FIFTH GENERATION COMPUTER SYSTEMS 1992

Edited by
Institute for New Generation
Computer Technology (ICOT)

Volume 1

Ohmsha, Ltd. IOS Press

FIFTH GENERATION COMPUTER SYSTEMS 1992

Edited by
Institute for New Generation
Computer Technology (ICOT)

Volume 1

Ohmsha, Ltd. IOS Press

FIFTH GENERATION COMPUTER SYSTEMS 1992

ISBN 4-274-07724-1 (Ohmsha)
ISBN 90-5199-099-5 (IOS Press)

Library of Congress Catalog Card Number: 92-073166

Published and distributed in *Japan* by
Ohmsha, Ltd.
3-1 Kanda Nishiki-cho, Chiyoda-ku, Tokyo 101, Japan

Distributed in *North America* by
IOS Press, Inc.
Postal Drawer 10558, Burke, VA 22009-0558, U.S.A.

United Kingdom by
IOS Press
73 Lime Walk, Headington, Oxford OX3 7AD, England

Europe and the rest of the world by
IOS Press
Van Diemenstraat 94, 1013 CN Amsterdam, Netherlands

Far East jointly by
Ohmsha, Ltd., IOS Press

Printed in Japan

FOREWORD

On behalf of the Organizing Committee, it is my great pleasure to welcome you to the International Conference on Fifth Generation Computer Systems 1992.

The Fifth Generation Computer Systems (FGCS) project was started in 1982 by the initiative of the late Professor Tohru Moto-Oka with the purpose of making a revolutionary new type of computers oriented to knowledge processing in the 1990s. After completing the initial and intermediate stages of research and development, we are now at the final point of our ten-year project and are rapidly approaching the completion of prototype Fifth Generation Computer Systems.

The research goals of the FGCS project were challenging, but we expect to meet most of them. We have developed a new paradigm of knowledge processing including the parallel logic language, KL1, and the parallel inference machine, PIM.

When we look back upon these ten years, we can find many research areas in knowledge processing related to this project, such as logic programming, parallel processing, natural language processing, and machine learning. Furthermore, there emerged many new applications of knowledge processing, such as legal reasoning and genetic information processing.

I believe that this new world of information processing will grow more and more in the future. When very large knowledge bases including common sense knowledge come out in full scale and are widely used, the knowledge processing paradigm will show its real power and will give us great rewards. From now on, we can enjoy fifth generation computer technology in many fields.

Following the same objective of creating such a new paradigm, there has been intense international collaboration, such as joint workshops with France, Italy, Sweden, the U.K., and the U.S.A., and joint research with U.S.A. and Swedish institutes on parallel processing applications.

Against this background,ICOT hosts the International Conference on Fifth Generation Computer Systems 1992 (FGCS'92). This is the last in a series of FGCS conferences; previous conferences were held in 1981, 1984 and 1988. The purpose of the conference is to present the final results of the FGCS project, as well as to promote the exchange of new ideas in the fields of knowledge processing, logic programming, and parallel processing.

FGCS'92 will take place over five days. The first two days will be devoted to the presentation of the latest results of the FGCS project, and will include invited lectures by leading researchers. The

remaining three days will be devoted to technical sessions for invited and submitted papers, the presentation of the results of detailed research done at ICOT, and panel discussions.

Professor D. Bjørner from the United Nations University, Professor J.A. Robinson from Syracuse University, and Professor C.A.R. Hoare from Oxford University kindly accepted our offer to give invited lectures.

Professor R. Kowalski from Imperial College is the chairperson of the plenary panel session on "A springboard for information processing in the 21st century." Professor Hajime Karatsu from Tokai University accepted our invitation to give a banquet speech.

During the conference, there will be demonstrations of the research results from the ten-year FGCS project. The Parallel Inference Machines and many kinds of parallel application programs will be highlighted to show the feasibility of the machines.

I hope that this conference will be a nice place to present all of the research results in this field up to this time, confirm the milestones, and propose a future direction for the research, development and applications of the fifth generation computers through vigorous discussions among attendees from all over the world. I hope all of the attendees will return to their own countries with great expectations in minds and feel that a new era of computer science has opened in terms of fifth generation computer systems.

Moreover, I wish that the friendship and frank cooperation among researchers from around the world, brewed in the process of fifth generation computer systems research, will grow and widen so that this small but strong relationship can help promote international collaboration for the brilliant future of mankind.

Hidehiko Tanaka
Conference Chairperson

FOREWORD

Esteemed guests, let me begin by welcoming you to the International Conference on Fifth Generation Computer Systems, 1992. I am Hideaki Kumano. I am the Director General of the Machinery and Information Industries Bureau of MITI.

We have been promoting the Fifth Generation Computer Systems Project, with the mission of international contributions to technological development by promoting the research and development of information technology in the basic research phase and distributing the achievements of that research worldwide. This international conference is thus of great importance in making our achievements available to all. It is, therefore, a great honor for me to be given the opportunity to make the keynote speech today.

1 Achievements of the Project

Since I took up my current post, I have had several opportunities to visit the project site. This made a great impression on me since it proved to me that Japanese technology can produce spectacular results in an area of highly advanced technology, covering the fields of parallel inference machine hardware and its basic software such as operating systems and programming languages; fields in which no one had any previous experience.

Furthermore, I caught a glimpse of the future use of fifth generation computer technology when I saw the results of its application to genetics and law. I was especially interested in the demonstration of the parallel legal inference system, since I have been engaged in the enactment and operation of laws at MITI. I now believe that the machines using the concepts of fifth generation computers will find practical applications in the enactment and operation of laws in the near future.

The research and development phase of our project will be completed by the end of this fiscal year. We will evaluate all the results. The committee for development of basic computer technology, comprised of distinguished members selected from a broad spectrum of fields, will make a formal evaluation of the project. This evaluation will take into account the opinions of those attending the conference, as well as the results of a questionnaire completed by overseas experts in each field. Even before this evaluation, however, I am convinced that the project has produced results that will have a great impact on future computer technology.

2 Features of the Fifth Generation Computer Systems Project

I will explain how we set our goals and developed a scheme that would achieve these high-level technological advances.

The commencement of the project coincided with the time when Japan was coming to be recognized as a major economic and technological power in the world community. Given these circumstances, the objectives of the project included not only the development of original and creative technology, but also the making of valuable international

contributions. In this regard, we selected a theme of "knowledge information processing", which would have a major impact on a wide area from technology through to the economy. The project took as its research goal the development of a parallel inference system, representing the paradigm of computer technology as applied to the theme.

The goal was particularly challenging at that time. I recalled the words of a participant at the first conference held in 1981. He commented that it was doubtful whether Japanese researchers could succeed in such a project since we, at that time, had very little experience in these fields.

However, despite the difficulties of the task ahead of us, we promoted the project from the viewpoint of contributing to the international community through research. In this regard, our endeavors in this area were targeted as pre-competitive technologies, namely basic research. This meant that we would have to start from scratch, assembling and training a group of researchers.

To achieve our goal of creating a paradigm of new computer technology, taking an integrated approach starting from basic research, we settled on a research scheme after exhaustive preliminary deliberations.

As part of its efforts to promote the dissemination of basic research results as international public assets, the government of Japan, reflecting its firm commitment to this area, decided to finance all research costs.

The Institute for New Generation Computer Technology (ICOT), the sponsor of this conference, was established to act as a central research laboratory where brainpower could be concentrated. Such an organization was considered essential to the development of an integrated technology that could be applied to both hardware and software. The Institute's research laboratory, that actually conducted the project's research and development, was founded precisely ten years ago, today, on June 1 of 1982. A number of highly qualified personnel, all of whom were excited by the ideal that the project pursued, were recruited from the government and industry. Furthermore, various ad hoc groups were formed to promote discussions among researchers in various fields, making ICOT the key center for research communication in this field.

The duration of the project was divided into three phases. Reviews were conducted at the end of each phase, from the viewpoint of human resources and technological advances, which made it possible to entrust various areas of the research. I believe that this approach increased efficiency, and also allowed flexibility by eliminating redundant areas of research.

We have also been heavily involved in international exchanges, with the aim of promoting international contributions. Currently, we are involved in five different international research collaboration projects. These include work in the theorem proving field with the Australian National University (ANU), and research into constraint logic programming with the Swedish Institute of Computer Science (SICS). The results of these two collaborations, on display in the demonstration hall, are excellent examples of what research collaboration can achieve. We have also promoted international exchange by holding international conferences and by hosting researchers from abroad at ICOT. And, we have gone to great lengths to make public our project's achievements, including in-

termediate results.

3 Succession of the Project's Ideal

This project is regarded as being the prototype for all subsequent projects to be sponsored by MITI.

It is largely due to the herculean efforts of the researchers, under the leadership of Dr. Fuchi and other excellent research leaders, that have led to the revolutionary advances being demonstrated at this conference.

In the light of these achievements, and with an eye to the future, I can now state that there is no question of the need to make international contributions the basis of the policies governing future technological development at MITI. This ideal will be passed on to all subsequent research and development projects.

A case in point is the Real World Computing (RWC) project scheduled to start this year. This project rests on a foundation of international cooperation. Indeed, the basic plan, approved by a committee a few days ago, specifically reflects the international exchange of opinions. The RWC project is a particularly challenging project that aims to investigate the fundamental principles of human-like flexible information processing and to implement it as a new information processing technology, taking full advantage of advancing hardware technologies. We will not fail to make every effort to achieve the project's objectives for use as common assets for all mankind.

4 International Response

As I mentioned earlier, I believe that the Fifth Generation Computer System Project has made valuable international contributions from its earliest stages. The project has stimulated international interest and responses from its outset. The great number of foreign participants present today illustrates this point.

Around the world, a number of projects received their initial impetus from our project: these include the Strategic Computing Initiative in the U.S.A., the EC's Esprit project, and the Alvey Project in the United Kingdom.

These projects were initially launched to compete with the Fifth Generation Computer Systems Project. Now, however, I strongly believe that since our ideal of international contributions has come to be understood around the globe, together with the realization that technology can not and should not be divided by borders, each project is providing the stimulus for the others, and all are making major contributions to the advancement of information processing technologies.

5 Free Access to the Project's Software

One of the great virtues of science, given an open environment, is the collaboration between researchers using a common base of technology.

Considering this, it would be impractical for one person or even one nation to attempt to cover the whole range of technological research and development. Therefore, the necessity of international cooperation is self-evident from the standpoint of advancing the human race as a whole.

In this vein, MITI has decided to promote technology globalism in the fields of science and technology, based on a concept of "international cooperative effort for creative activity and international exchange to maximize the total benefit of science and technology to mankind." We call this concept "techno-globalism".

It is also important to establish an environment based on "techno-globalism", that supports international collaboration in basic and original research as a resource to solve problems common to all mankind as well as the dissemination of the resulting achievements. This could be done through international cooperation.

To achieve this "techno-globalism" all countries should, as far as possible, allow free and easy access to their domestic technologies. This kind of openness requires the voluntary establishment of environments where anyone can access technological achievements freely, rather than merely asking other countries for information. It is this kind of international cooperation, with the efforts of both sides complementing each other, that can best accelerate the advancement of technology.

We at MITI have examined our policies from the viewpoint of promoting international technological advancement by using the technologies developed as part of this project, the superbness of which has encouraged us to set a new policy.

Our project's resources focused mainly on a variety of software, including parallel operating systems and parallel logic programming languages. To date, the results of such a national project, sponsored by the government, were available only for a fee and could be used only under various conditions once they became the property of the government. Therefore, generally speaking, although the results have been available to the public, in principle, they have not been available to be used freely and widely.

As I mentioned earlier, in the push toward reaching the goal of promoting international cooperation for technological advancement, Japan should take the initiative in creating an environment where all technologies developed in this project can be accessed easily. Now, I can formally announce that, concerning software copyrights in the research and development phase which are not the property of the government, the Institute for New Generation Computer Technology(ICOT), the owner of these copyrights of software products is now preparing to enable their free and and open use without charge.

The adoption of this policy not only allows anyone free access to the software technologies developed as part of the project, but also make it possible for interested parties to inherit the results of our research, to further advance the technology. I sincerely hope that our adopting this policy will maximize the utilization of researchers' abilities, and promote the advancement of the technologies of knowledge information processing and parallel processing, toward which all efforts have been concentrated during the project.

This means that our adopting this policy will not merely result in a one-way flow of technologies from Japan, but enhance the benefit to all mankind of the technological advancements brought on by a two-way flow of technology and the mutual benefits thus

obtained.

I should say that, from the outset of the Fifth Generation Computer Systems Project, we decided make international contributions an important objective of the project. We fashioned the project as the model for managing the MITI-sponsored research and development projects that were to follow. Now, as we near the completion of the project, we have decided to adopt a policy of free access to the software to inspire further international contributions to technological development.

I ask all of you to understand the message in this decision. I very much hope that the world's researchers will make effective use of the technologies resulting from the project and will devote themselves to further developing the technologies.

Finally, I'd like to close by expressing my heartfelt desire for this international conference to succeed in providing a productive forum for information exchange between participants and to act as a springboard for further advancements.

Thank you very much for bearing with me.

Hideaki Kumano
Director General
Machinery and Information Industries Bureau
Ministry of International Trade and Industry (MITI)

PREFACE

Ten years have passed since the FGCS project was launched with the support of the Japanese government. As soon as the FGCS project was announced it had a profound effect not only on computer scientists but also on the computer industry. Many countries recognized the importance of the FGCS project and some of them began their own similar national projects.

The FGCS project was initially planned as a ten-year project and this final fourth FGCS conference, therefore, has a historical meaning. For this reason the conference includes an ICOT session. The first volume contains a plenary session and the ICOT session. The plenary session is composed of many reports on the FGCS project with three invited lectures and a panel discussion.

In the ICOT session, the logic-based approach and parallel processing will be emphasized through concrete discussions. In addition to these, many demonstration programs have been prepared by ICOT at the conference site, the participants are invited to visit and discuss these exhibitions. Through the ICOT session and the exhibitions, the participants will understand clearly the aim and results of the FGCS project and receive a solid image of FGCS.

The second volume is devoted to the technical session which consists of three invited papers and technical papers submitted to this conference. Due to the time and space limitation of the conference, only 82 papers out of 256 submissions were selected by the program committee after careful and long discussion of many of the high quality papers submitted.

It is our hope that the conference program will prove to be both worthwhile and enjoyable. As a program chairperson, it is my great pleasure to acknowledge the support of a number of people. First of all, I would like to give my sincere thanks to the program committee members who put a lot of effort into making the program attractive. I owe much to the three program vice-chairpersons, Professor Makoto Amamiya, Dr. Shigeki Goto and Professor Fumio Mizoguchi. Many ICOT members, including Dr. Kazunori Ueda, Ken Satoh, Keiji Hirata, and Hideki Yasukawa have worked as key persons to organize the program. Dr. Koichi Furukawa, in particular, has played an indispensable role in overcoming many problems. I would also like to thank the many referees from many countries who replied quickly to the referees sheets.

Finally, I would like to thank the secretariat at ICOT, they made fantastic efforts to carry out the administrative tasks efficiently.

Hozumi Tanaka
Program Chairperson

CONFERENCE COMMITTEES

Steering Committee

Chairperson:	Kazuhiro Fuchi	ICOT
Members:	Hideo Aiso	Keio Univ.
	Setsuo Arikawa	Kyushu Univ.
	Ken Hirose	Waseda Univ.
	Takayasu Ito	Tohoku Univ.
	Hiroshi Kashiwagi	ETL
	Hajime Karatsu	Tokai Univ.
	Makoto Nagao	Kyoto Univ.
	Hiroki Nobukuni	NTT Data
	Iwao Toda	NTT
	Eiiti Wada	Univ. of Tokyo

Conference Committee

Chairperson:	Hidehiko Tanaka	Univ. of Tokyo
Vice-Chairperson:	Koichi Furukawa	ICOT
Members:	Makoto Amamiya	Kyushu Univ.
	Yuichiro Anzai	Keio Univ.
	Shigeki Goto	NTT
	Mitsuru Ishizuka	Univ. of Tokyo
	Kiyonori Konishi	NTT Data
	Takashi Kurozumi	ICOT
	Fumio Mizoguchi	Science Univ. of Tokyo
	Kunio Murakami	Kanagawa Univ.
	Sukeyoshi Sakai	ICOT(Chairperson, Management Committee)
	Masakazu Soga	ICOT(Chairperson, Technology Committee)
	Hozumi Tanaka	Tokyo Institute of Technology
	Shunichi Uchida	ICOT
	Kinko Yamamoto	JIPDEC
	Toshio Yokoi	EDR
	Akinori Yonezawa	Univ. of Tokyo
	Toshitsugu Yuba	ETL

Program Committee

Chairperson:	Hozumi Tanaka	Tokyo Institute of Technology
Vice-Chairpersons:	Makoto Amamiya	Kyushu Univ.
	Shigeki Goto	NTT
	Fumio Mizoguchi	Science Univ. of Tokyo
Members:	Koichi Furukawa	ICOT
	Kazunori Ueda	ICOT
	Ken Satoh	ICOT
	Keiji Hirata	ICOT
	Hideki Yasukawa	ICOT
	Hitoshi Aida	Univ. of Tokyo
	Yuichiro Anzai	Keio Univ.
	Arvind	MIT
	Ronald J. Brachman	AT&T
	John Conery	Univ. of Oregon
	Doug DeGroot	Texas Instruments
	Koichi Fukunaga	IBM Japan, Ltd.
	Jean-Luc Gaudiot	Univ. of Southern California
	Atsuhiro Goto	NTT
	Satoshi Goto	NEC Corp.
	Seif Haridi	SICS
	Ken'ichi Hagihara	Osaka Univ.

Makoto Haraguchi	Tokyo Institute of Technology
Ryuzo Hasegawa	ICOT
Hiromu Hayashi	Fujitsu Laboratories
Nobuyuki Ichiyoshi	ICOT
Mitsuru Ishizuka	Univ. of Tokyo
Tadashi Kanamori	Mitsubishi Electric Corp.
Yukio Kaneda	Kobe Univ.
Hirofumi Katsuno	NTT
Masaru Kitsuregawa	Univ. of Tokyo
Shigenobu Kobayashi	Tokyo Institute of Technology
Philip D. Laird	NASA
Catherine Lassez	IBM T.J. Watson
Giorgio Levi	Univ. di Pisa
John W. Lloyd	Univ. of Bristol
Yuji Matsumoto	Kyoto Univ.
Dale Miller	Univ. of Pennsylvania
Kuniaki Mukai	Keio Univ.
Hiroshi Motoda	Hitachi Ltd.
Katsuto Nakajima	Mitsubishi Electric Corp.
Ryohei Nakano	NTT
Kenji Nishida	ETL
Shojiro Nishio	Osaka Univ.
Stanley Peters	CSLI, Stanford Univ.
António Porto	Univ. Nova de Lisboa
Teodor C. Przymusinski	Univ. of California at Riverside
Vijay Saraswat	Xerox PARC
Taisuke Sato	ETL
Masahiko Sato	Tohoku Univ.
Heinz Schweppe	Institut für Informatik
Ehud Shapiro	The Weizmann Institute of Science
Etsuya Shibayama	Ryukoku Univ.
Kiyoshi Shibayama	Kyoto Univ.
Yoav Shoham	Stanford Univ.
Leon Sterling	Case Western Reserve Univ.
Mark E. Stickel	SRI International
Mamoru Sugie	Hitachi Ltd.
Akikazu Takeuchi	Sony CSL
Kazuo Taki	ICOT
Jiro Tanaka	Fujitsu Laboratories
Yuzuru Tanaka	Hokkaido Univ.
Philip Treleaven	University College, London
Syun Tutiya	Chiba Univ.
Shalom Tsur	MCC
D.H.D. Warren	Univ. of Bristol
Takahira Yamaguchi	Shizuoka Univ.
Kazumasa Yokota	ICOT
Minoru Yokota	NEC Corp.

Publicity Committee

Chairperson:	Kinko Yamamoto	JIPDEC
Vice-Chairperson:	Kunio Murakami	Kanagawa Univ.
Members:	Akira Aiba	ICOT
	Yuichi Tanaka	ICOT

Demonstration Committee

Chairperson:	Takashi Kurozumi	ICOT
Vice-Chairperson:	Shunichi Uchida	ICOT

LIST OF REFEREES

Abadi, Martin
Abramson, Harvey
Agha, Gul A.
Aiba, Akira
Aida, Hitoshi
Akama, Kiyoshi
Ali, Khayri A. M.
Alkalaj, Leon
Amamiya, Makoto
Amano, Hideharu
Amano, Shinya
America, Pierre
Anzai, Yuichiro
Aoyagi, Tatsuya
Apt, Krzysztof R.
Arikawa, Masatoshi
Arikawa, Setsuo
Arima, Jun
Arvind
Baba, Takanobu
Babaguchi, Noboru
Babb, Robert G., II
Bancilhon, François
Bansal, Arvind K.
Barklund, Jonas
Beaumont, Tony
Beeri, Catriel
Beldiceanu, Nicolas
Benhamou, Frederic R.
Bibel, Wolfgang
Bic, Lubomir
Biswas, Prasenjit
Blair, Howard A.
Boku, Taisuke
Bonnier, Staffan
Boose, John
Borning, Alan H.
Boutilier, Craig E.
Bowen, David
Brachman, Ronald J.
Bradfield, J. C.
Bratko, Ivan
Brazdil, Pavel
Briot, Jean-Pierre
Brogi, Antonio
Bruynooghe, Maurice
Bry, François
Bubst, S. A.
Buntine, Wray L.
Carlsson, Mats
Chikayama, Takashi
Chong, Chin Nyak
Chu, Lon-Chan
Ciepielewski, Andrzej
Clancey, William J.
Clark, Keith L.
Codish, Michael
Codognet, Christian
Conery, John
Consens, Mariano P.
Crawford, James M., Jr.
Culler, David E.
Dahl, Veronica
Davison, Andrew
de Bakker, Jaco W.
de Maindreville, Christophe
Debray, Saumya K.
Deen, S. M.
DeGroot, Doug
del Cerro, Luis Farinas
Demolombe, Robert
Denecker, Marc
Deransart, Pierre
Dincbas, Mehmet
Drabent, Wlodzimierz
Duncan, Timothy Jon
Dutra, Ines
Fahlman, Scott E.
Falaschi, Moreno
Faudemay, Pascal
Feigenbaum, Edward
Fitting, Melvin C.
Forbus, Kenneth D.
Fribourg, Laurent
Fujisaki, Tetsu
Fujita, Hiroshi
Fujita, Masayuki
Fukunaga, Koichi
Furukawa, Koichi
Gabbrielli, Maurizio
Gaines, Brian R.
Gardenfors, Peter
Gaudiot, Jean-Luc
Gazdar, Gerald
Gelfond, Michael
Gero, John S.
Giacobazzi, Roberto
Goebel, Randy G.
Goodwin, Scott D.
Goto, Atsuhiro
Goto, Satoshi
Goto, Shigeki
Grama, Ananth
Gregory, Steve
Gunji, Takao
Gupta, Anoop
Hagihara, Kenichi
Hagiya, Masami
Han, Jiawei
Hanks, Steve
Hara, Hirotaka
Harada, Taku
Haraguchi, Makoto
Haridi, Seif
Harland, James
Hasegawa, Ryuzo
Hasida, Kôiti
Hawley, David J.
Hayamizu, Satoru
Hayashi, Hiromu
Henry, Dana S.
Henschen, Lawrence J.
Herath, Jayantha
Hewitt, Carl E.
Hidaka, Yasuo
Higashida, Masanobu
Hiraga, Yuzuru
Hirata, Keiji
Hobbs, Jerry R.
Hogger, Christopher J.
Hong, Se June
Honiden, Shinichi
Hori, Koichi
Horita, Eiichi
Horiuchi, Kenji
Hsiang, Jieh
Iannucci, Robert A.
Ichikawa, Itaru

Ichiyoshi, Nobuyuki
Ida, Tetsuo
Ikeuchi, Katsushi
Inoue, Katsumi
Ishida, Toru
Ishizuka, Mitsuru
Iwasaki, Yumi
Iwayama, Makoto
Jaffar, Joxan
Jayaraman, Bharat
Kahn, Gilles
Kahn, Kenneth M.
Kakas, Antonios C.
Kameyama, Yukiyoshi
Kanade, Takeo
Kanamori, Tadashi
Kaneda, Yukio
Kaneko, Hiroshi
Kanellakis, Paris
Kaplan, Ronald M.
Kasahara, Hironori
Katagiri, Yasuhiro
Katsuno, Hirofumi
Kautz, Henry A.
Kawada, Tsutomu
Kawamura, Tadashi
Kawano, Hiroshi
Keller, Robert
Kemp, David
Kifer, Michael
Kim, Chinhyun
Kim, Hiecheol
Kim, WooYoung
Kimura, Yasunori
Kinoshita, Yoshiki
Kitsuregawa, Masaru
Kiyoki, Yasushi
Kluge, Werner E.
Kobayashi, Shigenobu
Kodratoff, Yves
Kohda, Youji
Koike, Hanpei
Komorowski, Jan
Konagaya, Akihiko
Kono, Shinji
Konolige, Kurt
Korsloot, Mark
Koseki, Yoshiyuki
Kraus, Sarit
Kumar, Vipin
Kunen, Kenneth
Kunifuji, Susumu
Kurita, Shohei
Kurokawa, Toshiaki
Kusalik, Anthony J.
Laird, Philip D.
Lassez, Catherine
Leblanc, Tom
Lescanne, Pierre
Leung, Ho-Fung
Levesque, Hector J.
Levi, Giorgio
Levy, Jean-Jacques
Lieberman, Henry A.
Lindstrom, Gary
Lloyd, John W.
Lusk, Ewing L.
Lytinen, Steven L.
Maher, Michael J.
Makinouchi, Akifumi
Manthey, Rainer
Marek, Victor
Marriott, Kim
Martelli, Maurizio
Maruoka, Akira
Maruyama, Fumihiro
Maruyama, Tsutomu
Masunaga, Yoshifumi
Matsubara, Hitoshi
Matsuda, Hideo
Matsumoto, Yuji
Matsuoka, Satoshi
McCune, William, W.
Memmi, Daniel
Mendelzon, Alberto O.
Menju, Satoshi
Meseguer, Jose
Michalski, Richard S.
Michie, Donald
Miller, Dale A.
Millroth, Håkan
Minami, Toshiro
Minker, Jack
Miyake, Nobuhisa
Miyano, Satoru
Miyazaki, Nobuyoshi
Miyazaki, Toshihiko
Mizoguchi, Fumio
Mizoguchi, Riichiro
Mori, Tatsunori
Morishita, Shinichi
Morita, Yukihiro
Motoda, Hiroshi
Mowtesi, Dawilo
Mukai, Kuniaki
Mukouchi, Yasuhito
Murakami, Kazuaki
Murakami, Masaki
Muraki, Kazunori
Muraoka, Yoichi
Nadathur, Gopalan
Naganuma, Jiro
Nagashima, Shigeo
Nakagawa, Hiroshi
Nakagawa, Takayuki
Nakajima, Katsuto
Nakamura, Junichi
Nakano, Miyuki
Nakano, Ryohei
Nakashima, Hideyuki
Nakashima, Hiroshi
Nakata, Toshiyuki
Nakayama, Masaya
Naqvi, Shamim A.
Natarajan, Venkat
Nikhil, Rishiyur, S.
Nilsson, Jørgen Fischer
Nilsson, Martin
Nishida, Kenji
Nishida, Toyoaki
Nishikawa, Hiroaki
Nishio, Shojiro
Nitta, Izumi
Nitta, Katsumi
Noyé, Jacques
Numao, Masayuki
Numaoka, Chisato
O'Rorke, Paul V.
Ogura, Takeshi
Ohki, Masaru
Ohmori, Kenji
Ohori, Atsushi
Ohsuga, Akihiko
Ohsuga, Setsuo
Ohwada, Hayato
Oka, Natsuki
Okumura, Manabu
Ono, Hiroakira
Ono, Satoshi
Overbeek, Ross A.

Oyanagi, Shigeru
Palamidessi, Catuscia
Panangaden, Prakash
Pearl, Judea
Pereira, Fernando C.
Pereira, Luís Moníz
Petrie, Charles J.
Plaisted, David A.
Plümer, Lutz
Poole, David
Popowich, Fred P.
Porto, António
Przymusinski, Teodor C.
Raina, Sanjay
Ramamohanarao, Kotagiri
Rao, Anand S.
Reddy, Uday S.
Ringwood, Graem A.
Robinson, John Alan
Rojas, Raul
Rokusawa, Kazuaki
Rossi, Francesca
Rossi, Gianfranco
Russell, Stuart J.
Sadri, Fariba
Saint-Dizier, Patrick
Sakai, Hiroshi
Sakai, Ko
Sakai, Shuichi
Sakakibara, Yasubumi
Sakama, Chiaki
Sakurai, Akito
Sakurai, Takafumi
Sangiorgi, Davide
Santos Costa, Vítor
Saraswat, Vijay A.
Sargeant, John
Sato, Masahiko
Sato, Taisuke
Sato, Yosuke
Satoh, Ken
Schweppe, Heinz
Seki, Hirohisa
Seligman, Jerry M.
Sergot, Marek J.
Sestito, Sabrina
Shanahan, Murray
Shapiro, Ehud
Shibayama, Etsuya
Shibayama, Kiyoshi
Shibayama, Shigeki
Shimada, Kentaro
Shin, Dongwook
Shinohara, Takeshi
Shintani, Toramatsu
Shoham, Yoav
Simonis, Helmut
Sirai, Hidetosi
Smith, Jan Magnus
Smolka, Gert
Sterling, Leon S.
Stickel, Mark E.
Stolfo, Salvatore. J.
Subrahmanian, V. S.
Sugano, Hiroyasu
Sugie, Mamoru
Sugiyama, Masahide
Sundararajan, Renga
Suwa, Masaki
Suzuki, Hiroyuki
Suzuki, Norihisa
Takagi, Toshihisa
Takahashi, Mitsuo
Takahashi, Naohisa
Takahashi, Yoshizo
Takayama, Yukihide
Takeda, Masayuki
Takeuchi, Akikazu
Takeuchi, Ikuo
Taki, Kazuo
Tamai, Tetsuo
Tamura, Naoyuki
Tanaka, Hozumi
Tanaka, Jiro
Tanaka, Katsumi
Tanaka, Yuzuru
Taniguchi, Rin-ichiro
Tatemura, Jun'ichi
Tatsuta, Makoto
Terano, Takao
Tick, Evan M.
Toda, Mitsuhiko
Togashi, Atsushi
Tojo, Satoshi
Tokunaga, Takenobu
Tomabechi, Hideto
Tomita, Shinji
Tomiyama, Tetsuo
Touretzky, David S.
Toyama, Yoshihito
Tsuda, Hiroshi
Tsur, Shalom
Tutiya, Syun
Uchihira, Naoshi
Ueda, Kazunori
Uehara, Kuniaki
Ueno, Haruki
van de Riet, Reinder P.
van Emden, Maarten H.
Van Hentenryck, Pascal
Van Roy, Peter L.
Vanneschi, Marco
Wada, Koichi
Wah, Benjamin W.
Walinsky, Clifford
Walker, David
Waltz, David L.
Warren, David H. D.
Warren, David Scott
Watanabe, Takao
Watanabe, Takuo
Watanabe, Toshinori
Watson, Ian
Watson, Paul
Weyhrauch, Richard W.
Wilk, Paul F.
Wolper, Pierre
Yamaguchi, Takahira
Yamamoto, Akihiro
Yamanaka, Kenjiroh
Yang, Rong
Yap, Roland
Yardeni, Eyal
Yasukawa, Hideki
Yokoo, Makoto
Yokota, Haruo
Yokota, Kazumasa
Yokota, Minoru
Yokoyama, Shoichi
Yonezaki, Naoki
Yonezawa, Akinori
Yoo, Namhoon
Yoon, Dae-Kyun
Yoshida, Hiroyuki
Yoshida, Kaoru
Yoshida, Kenichi
Yoshida, Norihiko
Yoshikawa, Masatoshi
Zerubia, Josiane B.

CONTENTS OF VOLUME 1

PLENARY SESSIONS

ICOT SESSIONS

CONTENTS OF VOLUME 2

FOUNDATIONS

AND-Parallelism and OR-Parallelism

Implementation Techniques

Extension of Logic Programming

Task Scheduling and Load Analysis

Concurrency

Databases and Distributed Systems

PLENARY SESSIONS

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Launching the New Era

Kazuhiro Fuchi

Director, Research Center
Institute for New Generation Computer Technology (ICOT)
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Thank you for coming to FGCS'92. As you know, we have been conducting a ten-year research project on fifth generation computer systems. Today is the tenth anniversary of the founding of our research center, making it exactly ten years since our project actually started.

The first objective of this international conference is to show what we have accomplished in our research during these ten years.

Another objective of this conference is to offer an opportunity for researchers to present the results of advanced research related to Fifth Generation Computer Systems and to exchange ideas. A variety of innovative studies, in addition to our own, are in progress in many parts of the world, addressing the future of computers and information processing technologies.

I constantly use the phrase "Parallel Inference" as the keywords to simply and precisely describe the technological goal of this project. Our hypothesis is that parallel inference technology will provide the core for those new technologies in the future—technologies that will be able to go beyond the framework of conventional computer technologies.

During these ten years I have tried to explain this idea whenever I have had the chance. One obvious reason why I have repeated the same thing so many times is that I wish its importance to be recognized by the public. However, I have another, less obvious, reason.

When this project started, an exaggerated image of the project was engendered, which seems to persist even now. For example, some people believed that we were trying, in this project, to solve in a mere ten years some of the most difficult problems in the field of artificial intelligence (AI), or to create a machine translation system equipped with the same capabilities as humans.

In those days, we had to face criticism, based upon that false image, that it was a reckless project trying to tackle impossible goals. Now we see criticism, from inside and outside the country, that the project has failed because it has been unable to realize those grand goals.

The reason why such an image was born appears to have something to do with FGCS'81—a conference we held one year before the project began. At that conference we discussed many different dreams and concepts. The substance of those discussions was reported as sensational news all over the world.

A vision with such ambitious goals, however, can never be materialized as a real project in its original form. Even if a project is started in accordance with the original form, it cannot be managed and operated within the framework of an effective research scheme. Actually, our plans had become much more modest by the time the project was launched.

For example, the development of application systems, such as a machine translation system, was removed from the list of goals. It is impossible to complete a highly intelligent system in ten years. A preliminary stage is required to enhance basic studies and to reform computer technology itself. We decided that we should focus our efforts on these foundational tasks. Another reason is that, at that time in Japan, some private companies had already begun to develop pragmatic, low-level machine-translation systems independently and in competition with each other.

Most of the research topics related to pattern recognition were also eliminated, because a national project called "Pattern Information Processing" had already been conducted by the Ministry of International Trade and Industry for ten years. We also found that the stage of the research did not match our own.

We thus deliberately eliminated most research topics covered by Pattern Information Processing from the scope of our FGCS project. However, those topics themselves are very important and thus remain major topics for research. They may become a main theme of another national project of Japan in the future.

Does all this mean that FGCS'81 was deceptive? I do not think so. First, in those days, a pessimistic outlook predominated concerning the future development of technological research. For example, there was a general trend that research into artificial intelligence would be of no practical use. In that sort of situation, there was considerable value in maintaining a positive attitude toward the future of technological research—whether this meant ten years or fifty. I believe that this was the very reason

why we received remarkable reactions, both positive and negative, from the public.

The second reason is that the key concept of Parallel Inference was presented in a clear-cut form at FGCS'81. Let me show you a diagram (Figure 1). This diagram is the one I used for my speech at FGCS'81, and is now a sort of "ancient document." Its draft was completed in 1980, but I had come up with the basic idea four years earlier. After discussing the concept with my colleagues for four years, I finally completed this diagram.

Here, you can clearly see our concept that our goal should be a "Parallel Inference Machine." We wanted to create an inference machine, starting with study on a variety of parallel architectures. For this purpose, research into a new language was necessary. We wanted to develop a 5G-kernel language—what we now call KL1. The diagram includes these hopes of ours.

The upper part of the diagram shows the research infrastructure. A personal inference machine or workstation for research purposes should be created, as well as a chip for the machine. We expected that the chip would be useful for our goal. The computer network should be consolidated to support the infrastructure. The software aspects are shown in the bottom part of the diagram. Starting with the study on software engineering and AI, we wanted to build a framework for high-level symbol processing, which should be used to achieve our goal. This is the concept I presented at the FGCS'81 conference.

I would appreciate it if you would compare this diagram with our plan and the results of the final stage of this project, when Deputy Director Kurozumi shows you them later. I would like you to compare the original structure conceived 12 years ago and the present results of the project so that you can appreciate what has been accomplished and criticize what is lacking or what was immature in the original idea.

Some people tend to make more of the conclusions drawn by a committee than the concepts and beliefs of an individual. It may sound a little bit beside point, but I have heard that there is a proverb in the West that goes, "The horse designed by a committee will turn out to be a camel."

The preparatory committee for this project had a series of enthusiastic discussions for three years before the project's launching. I thought that they were doing an exceptional job as a committee. Although the committee's work was great, however, I must say that the plan became a camel. It seems that their enthusiasm created some extra humps as well. Let me say in passing that some people seem to adhere to those humps. I am surprised that there is still such a so-called bureaucratic view even among academic people and journalists.

This is not the first time I have expressed this opinion of mine about the goal of the project. I have, at least in Japanese, been declaring it in public for the past ten years. I think I could have been discharged at any time had my opinion been inappropriate.

As the person in charge of this project, I have pushed forward with the lines of Parallel Inference based upon my own beliefs. Although I have been criticized as still being too ambitious, I have always been prepared to take responsibility for that.

Since the project is a national project, it goes without saying that it should not be controlled by one person. I have had many discussions with a variety of people for more than ten years. Fortunately, the idea of the project has not remained just a personal belief but has become a common belief shared by the many researchers and research leaders involved in the project.

Assuming that this project has proved to be successful, as I believe it has, this fact is probably the biggest reason for its success. For a research project to be successful, it needs to be favored by good external conditions. But the most important thing is that the research group involved has a common belief and a common will to reach its goals. I have been very fortunate to be able to realize and experience this over the past ten years.

So much for introductory remarks. I wish to outline, in terms of Parallel Inference, the results of our work conducted over these ten years. I believe that the remarkable feature of this project is that it focused upon one language and, based upon that language, experimented with the development of hardware and software on a large scale.

From the beginning, we envisaged that we would take logic programming and give it a role as a link that connects highly parallel machine architecture and the problems concerning applications and software. Our mission was to find a programming language for Parallel Inference.

A research group led by Deputy Director Furukawa was responsible for this work. As a result of their efforts, Ueda came up with a language model, GHC, at the beginning of the intermediate stage of the project. The two main precursors of it were Parlog and Concurrent Prolog. He enhanced and simplified them to make this model. Based upon GHC, Chikayama designed a programming language called KL1.

KL1, a language derived from the logic programming concept, provided a basis for the latter half of our project. Thus, all of our research plans in the final stage were integrated under a single language, KL1.

For example, we developed a hardware system, the Multi-PSI, at the end of the intermediate stage, and demonstrated it at FGCS'88. After the conference we made copies and have used them as the infrastructure for software research.

In the final stage, we made a few PIM prototypes, a Parallel Inference Machine that has been one of our final research goals on the hardware side. These prototypes are being demonstrated at this conference.

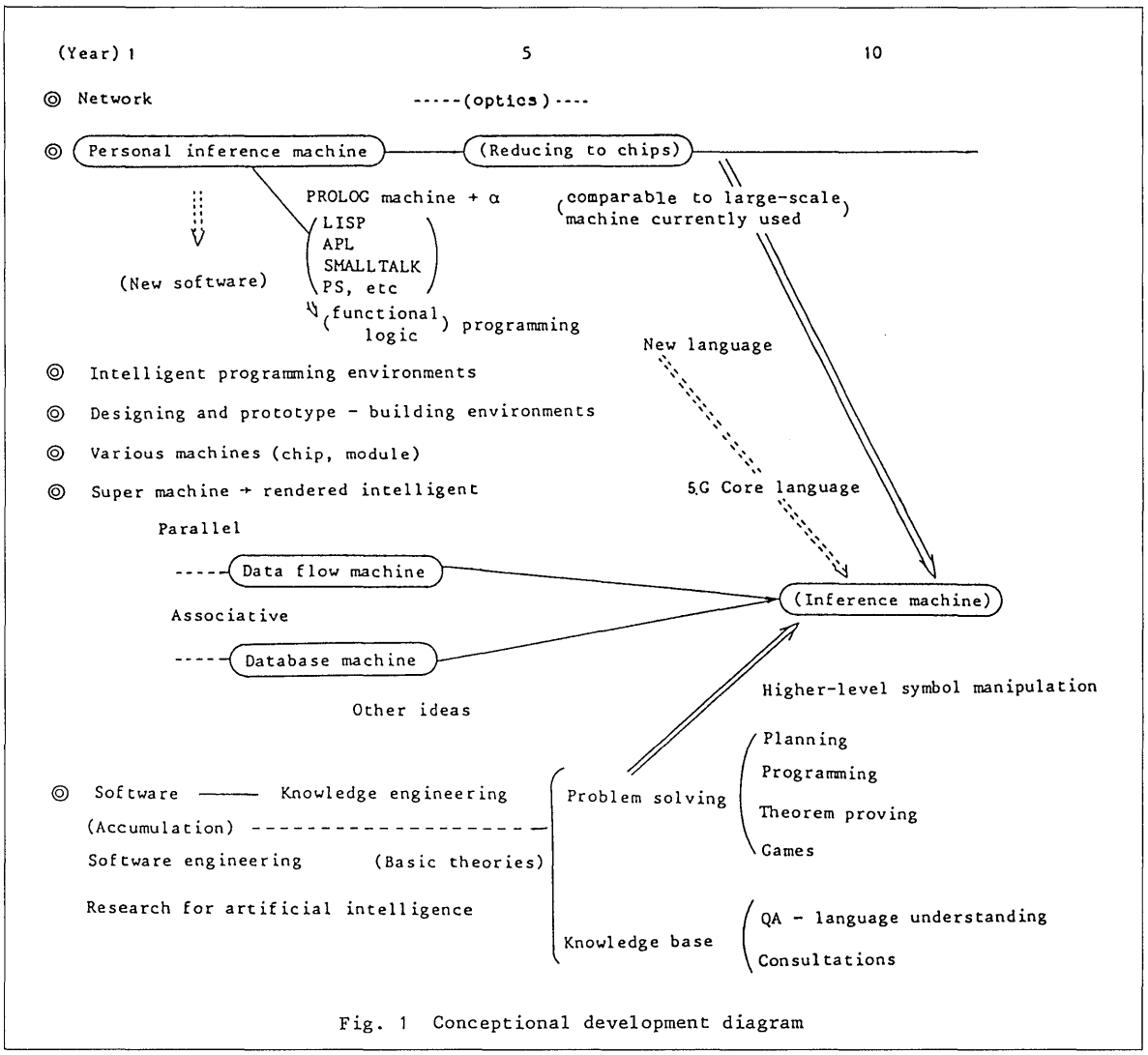


Fig. 1 Conceptional development diagram

T. Moto-oka (ed.): Fifth Generation Computer Systems (Proc. FGCS'81), JIPDEC: North-Holland, 1982, p. 113

Each prototype has a different architecture in its interconnection network and so forth, and the architecture itself is a subject of research. Viewed from the outside, however, all of them are KL1 machines.

Division Chief Uchida and Laboratory Chief Taki will show you details on PIM later. What I want to emphasize here is that all of these prototypes are designed, down to the level of internal chips, with the assumption that KL1, a language that could be categorized as a very high-level language, is a "machine language."

On the software side as well, our research topics were integrated under the KL1 language. All the application software, as well as the basic software such as operating systems, were to be written in KL1.

We demonstrated an operating system called PIMOS at FGCS'88, which was the first operating system software written in KL1. It was immature at that time, but has been improved since then. The full-fledged version of PIMOS now securely backs the demonstrations being shown at this conference.

Details will later be given by Laboratory Chief Chikayama, but I wish to emphasize that not only have we succeeded in writing software as complicated and huge as an operating system entirely in KL1, but we have also proved through our own experience that KL1 is much more appropriate than conventional languages for writing system software such as operating systems.

One of the major challenges in the final stage was to

demonstrate that KL1 is effective not only for basic software, such as operating systems and language implementations, but also for a variety of applications. As Laboratory Chief Nitta will report later, we have been able to demonstrate the effectiveness of KL1 for various applications including LSI-CAD, genetic analysis, and legal reasoning. These application systems address issues in the real world and have a virtually practical scale. But, again, what I wish to emphasize here is that the objective of those developments has been to demonstrate the effectiveness of Parallel Inference.

In fact, it was in the initial stage of our project that we first tried the approach of developing a project around one particular language. The technology was at the level of sequential processing, and we adopted ESP, an expanded version of Prolog, as a basis.

Assuming that ESP could play a role of KL0, our kernel language for sequential processing, a Personal Sequential Inference machine, called PSI, was designed as hardware. We decided to use the PSI machine as a workstation for our research. Some 500 PSIs, including modified versions, have so far been produced and used in the project.

SIMPOS, the operating system designed for PSI, is written solely in ESP. In those days, this was one of the largest programs written in a logic programming language.

Up to the intermediate stage of the project, we used PSI and SIMPOS as the infrastructure to conduct research on expert systems and natural language processing.

This kind of approach is indeed the dream of researchers, but some of you may be skeptical about our approach. Our project, though conducted on a large scale, is still considered basic research. Accordingly, it is supposed to be conducted in a free, unrestrained atmosphere so as to bring about innovative results. Some of you may wonder whether the policy of centering around one particular language restrains the freedom and diversity of research.

But this policy is also based upon my, or our, philosophy. I believe that research is a process of "assuming and verifying hypotheses." If this is true, the hypotheses must be as pure and clear as possible. If not, you cannot be sure of what you are trying to verify.

A practical system itself could include compromise or, to put it differently, flexibility to accommodate various needs. However, in a research project, the hypotheses must be clear and verifiable. Compromises and the like could be considered after basic research results have been obtained. This has been my policy from the very beginning, and that is the reason why I took a rather controversial or provocative approach.

We had a strong belief that our hypothesis of focusing on Parallel Inference and KL1 had sufficient scope for a world of rich and free research. Even if the hypothesis acted as a constraint, we believed that it would act as a creative constraint.

I would be a liar if I was to say that there was no resistance among our researchers when we decided upon the above policy. KL1 and parallel processing were a completely new world to everyone. It required a lot of courage to plunge headlong into this new world. But once the psychological barrier was overcome, the researchers set out to create new parallel programming techniques one after another.

People may not feel like using new programming languages such as KL1. Using established languages and systems only, or a kind of conservatism, seems to be the major trend today. In order to make a breakthrough into the future, however, we need a challenging and adventuring spirit. I think we have carried out our experiment with such a spirit throughout the ten-year project.

Among the many other results we obtained in the final stage was a fast theorem-proving system, or a prover. Details will be given in Laboratory Chief Hasegawa's report, but I think that this research will lead to the resurrection of theorem-proving research.

Conventionally, research into theorem proving by computers has been criticized by many mathematicians who insisted that only toy examples could be dealt with. However, very recently, we were able to solve a problem labelled by mathematicians as an 'open problem' using our prover, as a result of collaborative research with Australian National University.

The applications of our prover is not limited to mathematical theorem proving; it is also being used as the inference engine of our legal reasoning system. Thus, our prover is being used in the mathematics world on one hand, and the legal world on the other.

The research on programming languages has not ended with KL1. For example, a constraint logic programming language called GDCC has been developed as a higher-level language than KL1. We also have a language called Quixote.

From the beginning of this project, I have advocated the idea of integrating three types of languages—logic, functional, and object-oriented—and of integrating the worlds of programming and of databases. This idea has been materialized in the Quixote language; it can be called a deductive object-oriented database language.

Another language, CIL, was developed by Mukai in the study of natural language processing. CIL is a semantics representation language designed to be able to deal with situation theory. Quixote incorporates CIL in a natural form and therefore has the characteristics of a semantics representation language. As a whole, it shows one possible future form of knowledge representation languages.

More details on Quixote, along with the development of a distributed parallel database management system, Kappa-P, will be given by Laboratory Chief Yokota.

Thus far I have outlined, albeit briefly, the final results

of our ten-year project. Recalling what I envisaged ten years ago and what I have dreamed and hoped would materialize for 15 years, I believe that we have achieved as much as or more than what I expected, and I am quite satisfied.

Naturally, a national project is not performed for mere self-satisfaction. The original goal of this project was to create the core of next-generation computer technologies. Various elemental technologies are needed for future computers and information processing. Although it is impossible for this project alone to provide all of those technologies, we are proud to be able to say that we have created the core part, or at least provided an instance of it.

The results of this project, however, cannot be commercialized as soon as the project is finished, which is exactly why it was conducted as a national project. I estimate that it takes us another five years, which could be called a period for the "maturation of the technologies", for our results to actually take root in society. I had this prospect in mind when this project started ten years ago, and have kept declaring it in public right up until today. Now the project is nearing its end, but my idea is still the same.

There is often a gap of ten or twenty years between the basic research stage of a technology and the day it appears in the business world. Good examples are UNIX, C, and RISC, which has become popular in the current trend toward downsizing. They appear to be up-to-date in the business world, but research on them has been conducted for many years. The frank opinions of the researchers involved will be that industry has finally caught up with their research.

There is thus a substantial time lag between basic research and commercialization. Our project, from its very outset, set an eye on technologies for the far distant future. Today, the movement toward parallel computers is gaining momentum worldwide as a technology leading into the future. However, skepticism was dominant ten years ago. The situation was not very different even five years ago. When we tried to shift our focus on parallel processing after the initial stage of the project, there was a strong opinion that a parallel computer was not possible and that we should give it up and be happy with the successful results obtained in the initial stage.

In spite of the skepticism about parallel computers that still remains, the trend seems to be changing drastically. Thanks to constant progress in semiconductor technology, it is now becoming easier to connect five hundred, a thousand, or even more processor chips, as far as hardware technology is concerned.

Currently, the parallel computers that most people are interested in are supercomputers for scientific computation. The ideas there tend to still be vague regarding the software aspects. Nevertheless, a new age is dawning.

The software problem might not be too serious as long as scientific computation deals only with simple, scaled-up matrix calculations, but it will certainly become serious in the future. Now suppose this problem has been solved and we can nicely deal with all the aspects of large-scale problems with complicated overall structures. Then, we would have something like a general-purpose capability that is not limited to scientific computation. We might then be able to replace the mainframe computers we are using now.

The scenario mentioned above is one possibility leading to a new type of mainframe computer in the future. One could start by connecting a number of processor chips and face enormous difficulties with parallel software.

However, he or she could alternatively start by considering what technologies will be required in the future, and I suspect that the answer should be the Parallel Inference technology which we have been pursuing.

I am not going to press the above view upon you. However, I anticipate that if anybody starts research without knowing our ideas, or under a philosophy that he or she believes is quite different from ours, after many twists and turns that person will reach more or less the same concept as ours—possibly with small differences such as different terminology. In other words, my opinion is that there are not so many different essential technologies.

It may be valuable for researchers to struggle through a process of research independently from what has already been done, finally to find that they have followed the same course as somebody else. But a more efficient approach would be to build upon what has been done in this FGCS project and devote energy to moving forward from that point. I believe the results of this project will provide important insights for researchers who want to pursue general-purpose parallel computers.

This project will be finished at the end of this year. As for "maturation of the Parallel Inference technology", I think we will need a new form of research activities. There is a concept called "distributed cooperative computing" in the field of computation models. I expect that, in a similar spirit, the seeds generated in this project will spread both inside and outside the country and sprout in many different parts of the world.

For this to be realized, the results of this project must be freely accessible and available worldwide. In the software area, for example, this means that it is essential to disclose all our accomplishments including the source codes and to make them "international common public assets."

MITI Minister Watanabe and the Director General of the Bureau announced the policy that the results of our project could be utilized throughout the world. Enormous effort must have been made to formulate such a policy. I find it very impressive.

We have tried to encourage international collaboration for ten years in this project. As a result, we have

enjoyed opportunities to exchange ideas with many researchers involved in advanced studies in various parts of the world. They have given us much support and cooperation, without which this project could not have been completed.

In that regard, and also considering that this is a Japanese national project that aims at making a contribution, though it may only be small, toward the future of mankind, we believe that we are responsible for leaving our research accomplishments as a legacy to future generations and to the international community in a most suitable form. This is now realized, and I believe it is an important springboard for the future.

Although this project is about to end, the end is just another starting point. The advancement of computers and information processing technologies is closely related to the future of human society. Social thought, ideologies, and social systems that fail to recognize its significance will perish as we have seen in recent world history. We must advance into a new age now. To launch a new age, I fervently hope that the circle of those who share our passion for a bright future will continue to expand. Thank you.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Overview of the Ten Years of the FGCS Project

Takashi Kurozumi

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Kurozumi @ icot.or.jp

Abstract

This paper introduces how the FGCS Project started, its overall activities and the results of the FGCS project. The FGCS Project was launched in 1982 after a three year preliminary study stage. The basic framework of the fifth generation computer is parallel processing and inference processing based on logic programming. Fifth generation computers were viewed as suitable for the knowledge information processing needs of the near future. ICOT was established to promote the FGCS Project. This paper shows not only, ICOT's efforts in promoting the FGCS project, but relationship between ICOT and related organizations as well. I, also, conjecture on the parallel inference machines of the near future.

1 Preliminary Study Stage for the FGCS Project

The circumstances prevailing during the preliminary stage of the FGCS Project, from 1979 to 1981, can be summarized as follows.

·Japanese computer technologies had reached the level of the most up-to-date overseas computer technologies.

·A change of the role of the Japanese national project for computer technologies was being discussed whereby there would be a move away from improvement of industrial competitiveness by catching up with the latest European computer technologies and toward world-wide scientific contribution through the risky development of leading computer technologies.

In this situation, the Japanese Ministry of International Trade and Industry (MITI) started study on a new project - the Fifth Generation Computer Project. This term expressed MITI's will to develop leading technologies that would progress beyond the fourth generation computers due to appear in the near future and which would anticipate upcoming trends.

The Fifth Generation Computer Research Committee and its subcommittee (Figure 1-1) were established in 1979. It took until the end of 1981 to decide on target technologies and a framework for the project.

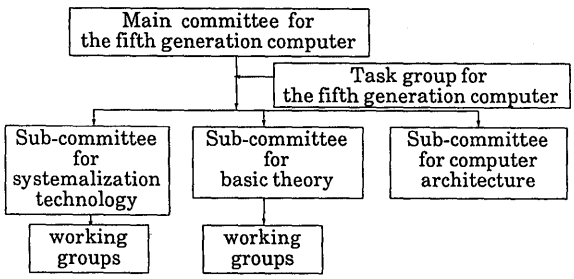


Figure 1-1 Organization of the Fifth Generation Computer Committee

Well over one hundred meetings were held with a similar number of committee members participating. The following important near-future computer technologies were discussed.

- Inference computer technologies for knowledge processing
- Computer technologies to process large-scale data bases and knowledge bases
- High performance workstation technologies
- Distributed functional computer technologies
- Super-computer technologies for scientific calculation

These computer technologies were investigated and discussed from the standpoints of international contribution by developing original Japanese technologies, the important technologies in future, social needs and conformance with Japanese governmental policy for the national project.

Through these studies and discussions, the committee decided on the objectives of the project by

the end of 1980, and continued future studies of technical matters, social impact, and project schemes.

The committee's proposals for the FGCS Project are summarized as follows.

① The concept of the Fifth Generation Computer: To have parallel (non-Von Neumann) processing and inference processing using knowledge bases as basic mechanisms. In order to have these mechanisms, the hardware and software interface is to be a logic program language (Figure 1-2).

② The objectives of the FGCS project: To develop these innovative computers, capable of knowledge information processing and to overcome the technical restrictions of conventional computers.

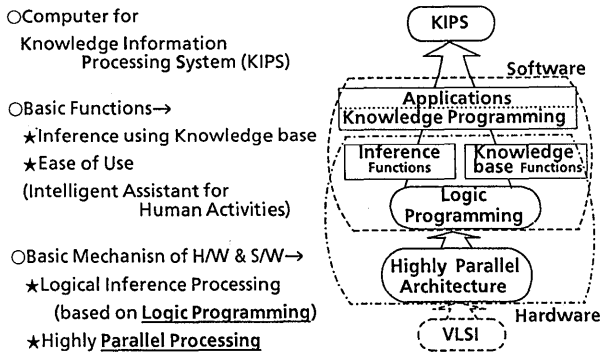


Figure 1-2 Concept of the Fifth Generation Computer

③ The goals of the FGCS project: To research and develop a set of hardware and software technologies for FGCS, and to develop an FGCS prototype system consisting of a thousand element processors with inference execution speeds of between 100M LIPS and 1G LIPS (Logical Inferences Per Second).

④ R&D period for the project: Estimated to be 10 years, divided into three stages.

- 3-year initial stage for R&D of basic technologies
- 4-year intermediate stage for R&D of subsystems
- 3-year final stage for R&D of total prototype system

MITI decided to launch the Fifth Generation Computer System (FGCS) project as a national project for new information processing, and made efforts to acquire a budget for the project.

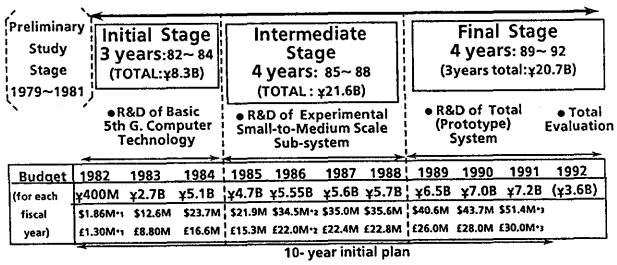
At the same time, the international conference on FGCS '81 was prepared and held in October 1981 to announce these results and to hold discussions on the topic with foreign researchers.

2 Overview of R&D Activities and Results of the FGCS Project

2.1 Stages and Budgeting in the FGCS Project

The FGCS project was designed to investigate a large number of unknown technologies that were yet to be developed. Since this involved a number of risky goals, the project was scheduled over a relatively long period of ten years. This ten-year period was divided into three stages.

- In the initial stage (fiscal 1982-1984), the purpose of R&D was to develop the basic computer technologies needed to achieve the goal.
- In the intermediate stage (fiscal 1985-1988), the purpose of R&D was to develop small to medium subsystems.
- In the final stage (fiscal 1989-1992), the purpose of R&D was to develop a total prototype system. The final stage was initially planned to be three years. After reexamination halfway through the final stage, this stage was extended to four years to allow evaluation and improvement of the total system in fiscal year 1992. Consequently, the total length of this project has been extended to 11 years.



● R&D are carried out under the auspices of MITI.
(All budget (Total budgets:¥54.6B) are covered by MITI.)

*1 $1 = ¥215, £1 = ¥307 (1982~1985)
*2 $1 = ¥160, £1 = ¥250 (1986~1990)
*3 $1 = ¥140, £1 = ¥240 (1991~)

Figure 2-1 Budgets for the FGCS project

Each year the budget for the following years R&D activities was decided. MITI made great efforts in negotiating each year's budget with the Ministry of Finance. The budgets for each year, which are all covered by MITI, are shown in Figure 2-1. The total budget for the 3-year initial stage was about 8 billion yen. For the 4-year intermediate stage, it was about 22 billion yen. The total budget for 1989 to 1991 was around 21 billion yen. The budget for 1992 is estimated to be 3.6 billion yen.

Consequently, the total budget for the 11-year period of the project will be about 54 billion yen.

2.2 R&D subjects of each stage

At the beginning, it was considered that a detailed R&D plan could not be decided in detail for a period as long as ten years. The R&D goals and the means to reach these goals were not decided in detail. During the project, goals were sought and methods decided by referring back to the initial plan at the beginning of each stage.

The R&D subjects for each stage, shown in Figure 2-2, were decided by considering the framework and conditions mentioned below.

We defined 3 groups of 9 R&D subjects at the beginning of the initial stage by analyzing and rearranging the 5 groups of 10 R&D subjects proposed by the Fifth Generation Computer Committee.

At the end of the initial stage, the basic research themes of machine translation and speech, figure and image processing were excluded from this project. These were excluded because computer vender efforts on these technologies were recognized as having become very active.

In the middle of the intermediate stage, the task of developing a large scale electronic dictionary was transferred to EDR (Electronic Dictionary Research Center), and development of CESP (Common ESP system on UNIX) was started by AIR (AI language Research Center).

The basic R&D framework for promoting this project is to have common utilization of developed software by unifying the software development environment (especially by unifying programming languages). By utilizing software development systems and tools, the results of R&D can be evaluated and improved. Of course, considering the nature of this project, there is another reason making it difficult or impossible to use commercial products as a software development environment.

In each stage, the languages and the software development environment are unified as follows.

- Initial stage: Prolog on DEC machine
- Intermediate stage: ESP on PSI and SIMPOS
- Final stage: KL1 on Multi-PSI (or PIM) and PIMOS (PSI machines are also used as pseudo multi-PSI systems.) (Figure 2-6)

2.3 Overview of R&D Results of Hardware System

Hardware system R&D was carried out by the subjects listed listed below in each stage.

① Initial stage

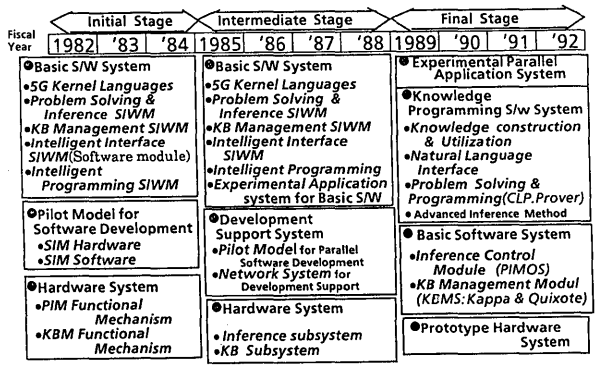


Figure 2-2 Transition of R&D subjects in each stage

ⓐ Functional mechanism modules and simulators for PIM (Parallel Inference Machine) of the hardware system

ⓑ Functional mechanism modules and simulators for KBM (Knowledge Base Machine) of the hardware system

ⓒ SIM (Sequential Inference Machine) hardware of pilot model for software development

② Intermediate Stage

ⓐ Inference subsystem of the hardware system.

ⓑ Knowledge base subsystem of the hardware system

ⓒ Pilot model for parallel software development of the development support system.

③ Final Stage

ⓐ Prototype hardware system

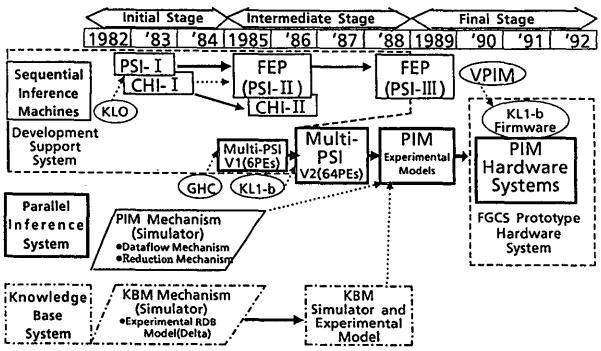


Figure 2-3 Transition of R&D results of Hardware System

The major R&D results on SIM were the PSI (Personal Sequential Inference Machine) and CHI (high performance back-end inference unit). In the initial stage, PSI- I (① ⓒ) was developed as KL0 (Kernel Language Version 0) machine. PSI- I had

around 35 KLIPS (Logical Inference Per Second) execution speed. Around 100 PSI-Ⅰ machines were used as main WSs (workstations) for the sequential logic programming language, ESP, in the first half of the intermediate stage. CHI-Ⅰ (① ⓒ) showed around 200 KLIPS execution speed by using WAM instruction set and high-speed devices. In the intermediate stage, PSI was redesigned as multi-PSI FEP (Front End Processor) and PSI-Ⅱ, and has performance of around 330-400 KLIPS. CHI was also redesigned as CHI-Ⅱ (② ⓒ), with more than 400 KLIPS performance. PSI-Ⅱ machines were the main WSs for ESP after the middle of the intermediate stage, and were able to be used for KL1 by the last year of the intermediate stage. PSI-Ⅲ was developed as a commercial product by a computer company by using PIM/m CPU technologies, with the permission of MITI, and by using UNIX.

R&D on PIM continued throughout the project, as follows.

- In the initial stage, experimental PIM hardware simulators and software simulators with 8 to 16 processors were trial-fabricated based on data flow and reduction mechanisms (③ⓐ).
- In the intermediate stage, we developed multi-PSI V1, which was to construct 6 PSI-Is, as the first version of the KL1 machine. The performance of this machine was only several KLIPS because of the KL1 emulator (② ⓒ). It did, however, provide evaluation and experience by developing a very small parallel OS in KL1. This meant that we could develop multi-PSI V2 with 64 PSI-Ⅱ CPUs connected by a mesh network (② ⓐ). The performance of each CPU for KL1 was around 150 KLIPS, and the average performance of the full multi-PSI V2 was 5 MLIPS. This speed was enough to significantly improved to encourage efforts to develop various parallel KL1 software programs including an practical OS.
- After development of multi-PSI V2, we promote the design (② ⓐ) and trial-fabrication of PIM experimental models (③ⓐ).
- At present, we are completing development of prototype hardware consisting of 3 large scale PIM modules and 2 small scale experimental PIM modules (③ ⓐ). These PIM modules are designed to be equally suited to the KL1 machine for inference and knowledge base management, and to be able to be installed all programs written by KL1. This is in spite of their using different architecture.

The VPIM system is a KL1-b language processing system which gives a common base for PIM firmware for KL1-b developed on conventional computers.

R&D on KBM continued until the end of the intermediate stage. An experimental relational data base machine (Delta) with 4 relational algebraic engines was trial-fabricated in the initial stage (① ⓑ). During the intermediate stage, a deductive data base simulator was developed to use PSIs with an accelerator for comparison and searching. An experimental system was also developed with multiple-multiple name spaces, by using CHI. Lastly, a knowledge base hardware simulator with unification engines and multi-port page memory was developed in this stage (② ⓑ). We developed DB/KB management software, called Kappa, on concurrent basic software themes. At the beginning of the final stage, we thought that adaptability of PIM with Kappa for the various description forms for the knowledge base was more important than effectivity of KBM with special mechanism for the specific KB forms. In other words, we thought that deductive object-oriented DB technologies was not yet matured to design KBM as a part of the prototype system.

2.4 Overview of R&D Results of Software Systems

The R&D of software systems was carried out by a number of subjects listed below in each stage.

① Initial stage
- Basic software
  - ⓐ 5G Kernel Languages
  - ⓑ Problem solving and inference software module
  - ⓒ Knowledge base management software module
  - ⓓ Intelligent interface software module
  - ⓔ Intelligent programming software module
  - ⓕ SIM software of pilot model for development support

② Basic software system in the intermediate stage
- ⓐ-ⓔ (as in the initial stage)
- ⓕ Experimental application system for basic software module

③ Final stage
- Basic software system
  - ⓐ Inference Control module
  - ⓑ KB management module
- Knowledge programming software
  - ⓒ Problem solving and programming module
  - ⓓ Natural language interface module
  - ⓔ Knowledge construction and utilization module
  - ⓕ Advanced problem solving inference method

ⓖ Experimental parallel application system

To make the R&D results easy to understand, I will separate the results for languages, basic software, knowledge programming and application software.

2.4.1 R&D results of Fifth Generation Computer languages

As the first step in 5G language development, we designed sequential logic programming languages KL0 and ESP (Extended Self-contained Prolog) and developed these language processors (① ⓐ). KL0, designed for the PSI hardware system, is based on Prolog. ESP has extended modular programming functions to KL0 and is designed to describe large scale software such as SIMPOS and application systems.

As a result of research on parallel logic programming language, Guarded Horn Clauses, or GHC, was proposed as the basic specification for KL1 (Kernel Language Version 1) (①ⓐ). KL1 was, then, designed by adding various functions to KL1 such as a macro description (②ⓐ). KL1 consists of a machine level language (KL1-b (base)), a core language (KL1-c) for writing parallel software and pragma (KL1-p) to describe the division of parallel processes. Parallel inference machines, multi-PSI and PIM, are based on KL1-b. Various parallel software, including PIMOS, is written in KL1-c and KL1-p.

A'um is an object oriented language. The results of developing the A'um experimental language processor reflect improvements in KL1 (②ⓐ, ③ⓐ).

To research higher level languages, several languages were developed to aid description of specific research fields. CIL (Complex Indeterminate Language) is the extended language of Prolog that describes meanings and situations for natural language processing (① ⓓ, ② ⓓ). CRL (Complex Record Language) was developed as a knowledge representation language to be used internally for deductive databases on nested relational DB software (② ⓒ). CAL (Contrainte Avec Logique) is a sequential constraint logic language for constraint programming (②ⓑ).

Mandala was proposed as a knowledge representation language for parallel processing, but was not adopted because it lacks a parallel processing environment and we had enough experience with it in the initial stage (①ⓒ).

Quixote is designed as a knowledge representation language and knowledge-base language for parallel processing based on the results of evaluation by CIL and CRL. Quixote is also a deductive object-oriented database language and play the key role in KBMS. A language processor is currently being developed for Quixote. GDCC(Guarded Definite Clause with Constraints)

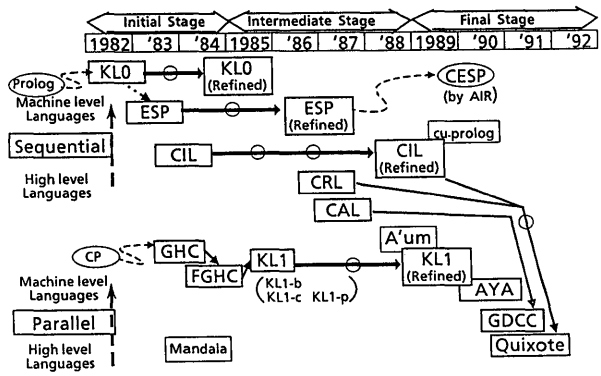


Figure 2-4 Transition of R&D of 5G Languages

is a parallel constraint logic language that processes CAL results.

2.4.2 R&D Results of Basic Software (OS)

In the initial stage, we developed a preliminary programming and operating system for PSI, called SIMPOS, using ESP (① ⓔ ⓕ). We continued to improve SIMPOS by adding functions corresponding to evaluation results. We also took into account the opinions of inside users who had developed software for the PSI machine using SIMPOS (②ⓑⓕ).

Since no precedent parallel OS which is suited for our aims had been developed anywhere in the world, we started to study parallel OS using our experiences of SIMPOS development in the initial stage. A small experimental PIMOS was developed on the multi-PSI V1 system in the first half of the intermediate stage (②ⓑ). Then, the first version of PIMOS was developed on the multi-PSI V2 system, and was used by KL1 users (② ⓑ). PIMOS continued to be improved by the addition of functions such as remote access, file access and debugging support (③ⓐ).

The Program Development Support System was also developed by the end of the intermediate stage (②ⓑ).

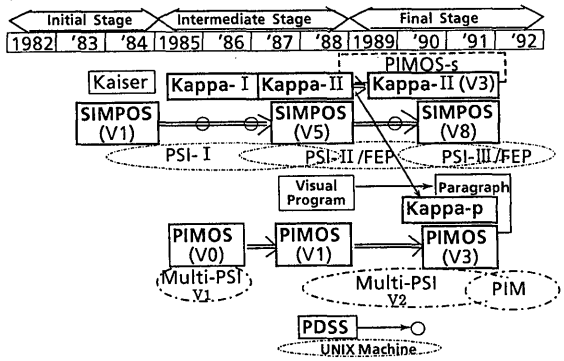


Figure 2-5 Transition of basic software R&D

Paragraph was developed as a parallel programming support system for improving concurrency and load distribution by the indication results of parallel processing (③ⓐ).

In regard to DB/KB management software, Kaiser was developed as a experimental relational DB management software in the initial stage (① ⓒ). Then, Kappa-Ⅰ and Kappa-Ⅱ were developed to provide the construction functions required to build a large scale DB/KB that could be used for natural language processing, theorem proving and various expert systems (② ⓒ). Kappa-Ⅰ and Kappa-Ⅱ,based on nested relational model, are aimed at the database engine of deductive object-oriented DBMS.

Recently, a parallel version of Kappa , Kappa-P, is being developed. Kappa-P can manage distributed data bases stored on distributed disks in PIM. (③ ⓑ) Kappa-P and Quixote constitute the KBMS.

2.4.3 R&D Results of Problem Solving and Programming Technologies

Throughout this project, from the viewpoint of similarity mathematical theorem proving and program specification, we have been investigating proving technologies. The CAP (Computer Aided Proof) system was experimentally developed in the initial stage (② ⓔ). TRS (Term Rewriting System) and Metis were also developed to support specific mathematical reasoning, that is, the inference associated equals sign (②ⓔ).

An experimental program for program verification and composition, Argus, was developed by the end of the intermediate stage (① ⓔ and ② ⓔ). These research themes concentrated on R&D into the MGTP theorem prover in the final stage(③ⓒ).

Meta-programming technologies, partial evaluation technologies and the learning mechanism were investigated as basic research on advanced problem solving and the inference method (①ⓑ, ②ⓑ, ③ⓕ).

2.4.4 R&D Results on Natural Language Processing Technologies

Natural language processing tools such as BUP (Bottom-Up Parser) and a miniature electronic dictionary were experimentally developed in the initial stage (① ⓓ). These tools were extended, improved and arranged into LTB (Language Tool Box). LTB is a library of Japanese processing software modules such as LAX (Lexical Analyzer), SAX (Syntactic Analyzer), a text generator and language data bases (②ⓓ, ③ⓓ).

An experimental discourse understanding system, DUALS, was implemented to investigate context processing and semantic analysis using these language processing tools (① ⓓ,② ⓓ). An experimental argument system, called Dulcinia, is being implemented in the final stage (③ⓓ).

2.4.5 R&D Results on Knowledge Utilization Technologies and Experimental Application Systems

In the intermediate stage we implemented experimental knowledge utilization tools such as APRICOT, based on hypothetical reasoning technology, and Qupras, based on qualitative reasoning technology (② ⓒ). At present, we are investigating such inference mechanisms for expert systems as assumption based reasoning and case based reasoning, and implementing these as knowledge utilization tools to be applied to the experimental application system (③ⓔ).

As an application system, we developed, in Prolog, an experimental CAD system for logic circuit design support and wiring support in the initial stage. We also developed several experimental expert systems such as a CAD system for layout and logic circuit design, a troubleshooting system, a plant control system and a go-playing system written in ESP (②ⓕ, etc.).

Small to medium parallel programs written in KL1 were also developed to test and evaluate parallel systems by the end of the intermediate stage. These were improved for application to PIM in the final stage. These programs are PAX (a parallel semantics analyzer), Pentomino solver, shortest path solver and Tsume-go.

We developed several experimental parallel systems, implemented using KL1 in the final stage, such as LSI-CAD system (for logical simulation, wire routing, block layout, logical circuit design), genetic information processing system, legal inference system based on case based reasoning, expert systems for troubleshooting, plant control and go-playing (3g).

Some of these experimental systems were developed from other earlier sequential systems in the intermediate stage while others are new application fields that started in the final stage.

2.5 Infrastructure of the FGCS Project

As explained in 2.2, the main language used for software implementation in the initial stage was Prolog. In the intermediate stage, ESP was mainly used, and in the final stage KL1 was the principle language.

Therefore, we used a Prolog processing system on a conventional computer and terminals in the initial stage. SIMPOS on PSI (Ⅰ and Ⅱ) was used as the workbench for sequential programming in

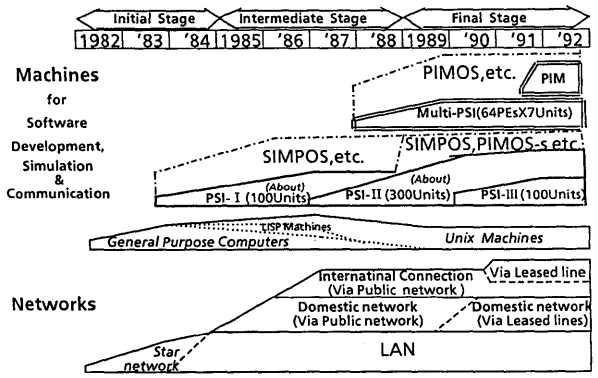


Figure 2-6 Infrastructure for R&D

the intermediate stage. We are using PSI (II and III) as a workbench and remote terminals to parallel machines (multi-PSIs and PIMs) for parallel programming in the final stage. We have also used conventional machines for simulation to design PIM and a communication (E-mail, etc.) system.

In regard to the computer network system, LAN has been used as the in-house system, and LAN has been connected to domestic and international networks via gateway systems.

3 Promoting Organization of the FGCS Project

ICOT was established in 1982 as a non-profit core organization for promoting this project and it began R&D work on fifth generation computers in June 1982, under the auspices of MITI.

Establishment of ICOT was decided by considering the following necessity and effectiveness of a centralized core research center for promoting originative R&D,

- R&D themes should be directed and selected by powerful leadership, in consideration of hardware and software integration, based on a unified framework of fifth generation computers, throughout the ten-year project period.
- It was necessary to develop and nurture researchers working together because of the lack of researchers in this research field.
- A core center was needed to exchange information and to collaborate with other organizations and outside researchers.

ICOT consists of a general affairs office and a research center (Figure 3-1).

The organization of the ICOT research center was changed flexibly depending on the progress being made. In the initial stage, the research center consisted of a research planning department and three research laboratories. The number of

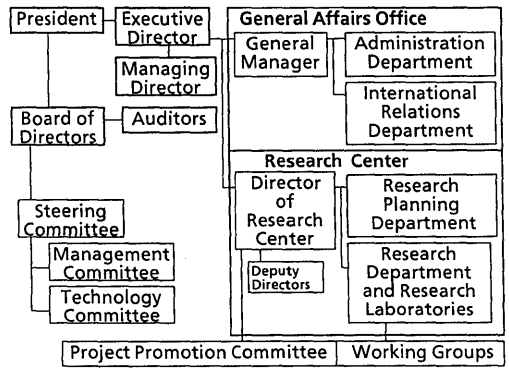


Figure 3-1 ICOT Organization

laboratories was increased to five at the beginning of the intermediate stage. These laboratories became one research department and seven laboratories in 1990.

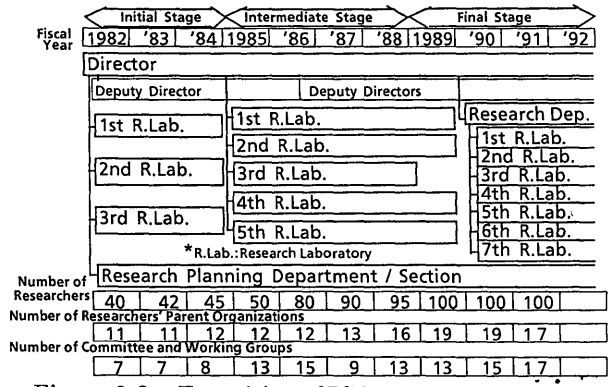


Figure 3-2 Transition of ICOT research center organization

The number of researchers at the ICOT research center has increased yearly, from 40 in 1982 to 100 at the end of the intermediate stage.

All researchers at the ICOT research center have been transferred from national research centers, public organizations, and computer vendors, and the like. To encourage young creative researchers and promote originative R&D, the age of dispatched researchers is limited to 35 years old. Because all researchers are normally dispatched to the ICOT research center for three to four years, ICOT had to receive and nurture newly transferred researchers. We must make considerable effort to continue to consistently lead R&D in the fifth generation computer field despite researcher rotation. This rotation has meant that we were able to maintain a staff of researchers in their 30's, and also could easily change the structure of organization in the ICOT research center.

In total, 184 researchers have been transferred to

the ICOT research center with an average transfer period of 3 years and eight months (including around half of the dispatched researchers who are presently at ICOT).

The number of organizations which dispatched researchers to ICOT also increased, from 11 to 19. This increase in participating organizations was caused by an expanding scheme of the supporting companies, around 30 companies, to dispatch researchers to ICOT midway through the intermediate stage.

The themes each laboratory was responsible for changed occasionally depending on the progress being made.

Figure 3-3 shows the present assignment of research themes to each research laboratory.

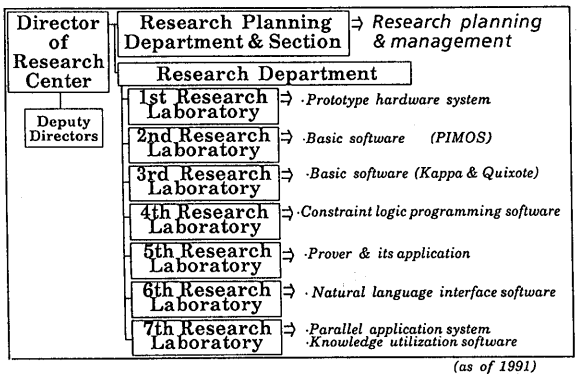


Figure 3-3 ICOT research center organization

Every year we invited several visiting researchers from abroad for several weeks at ICOT's expense to discuss and to exchange opinion on specific research themes with ICOT researchers. Up to the present, we have invited 74 researchers from 12 countries in this program.

We also received six long-term (about one year each) visiting researchers from foreign governmental organizations based on memorandums with the National Science Foundation (NSF) in the United States, the Institute National de Recherche en Informatique et Automatiqeu (INRIA) in France, and the Department of Trade and Industry (DTI) in the United Kingdom (Figures 3-2 and 3-4).

Figure 3-4 shows the overall structure for promoting this project. The entire cost for the R&D activities of this project is supported by MITI based on the entrust contract between MITI and ICOT. Yearly and at the beginning of each stage we negotiate our R&D plan with MITI. MITI receives advice of this R&D plan and evaluations of R&D results and ICOT research activities from the FGCS project advisory committee.

ICOT executes the core part of R&D and has contracts with eight computer companies for

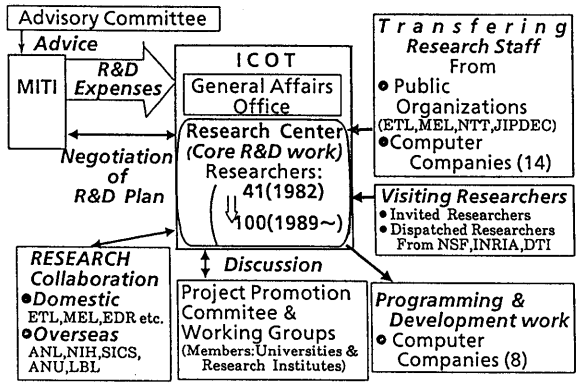


Figure 3-4 Structure for promoting FGCS project

experimental production of hardware and developmental software. Consequently, ICOT can handle all R&D activities, including the developmental work of computer companies towards the goals of this project.

ICOT has set up committee and working groups to discuss and to exchange opinions on overall plans results and specific research themes with researchers and research leaders from universities and other research institutes. Of course, construction and the themes of working groups are changed depending on research progress. The number of people in a working group is around 10 to 20 members, so the total number in the committee and working groups is about 150 to 250 each year.

Another program for information exchange and collaborative research activities and diffusion of research results will be described in the following chapter.

4 Distribution of R&D Results and International Exchange Activities

Because this project is a national project in which world-wide scientific contribution is very important, we have made every effort to include our R&D ideas, processes and project results when presenting ICOT activities. We, also, collaborate with outside researchers and other research organizations.

We believe these efforts have contributed to progress in parallel and knowledge processing computer technologies. I feel that the R&D efforts in these fields have increased because of the stimulative effect of this project. We hope that R&D efforts will continue to increase through distribution of this projects R&D results. I believe that many outside researchers have also made significant contributions to this project through

their discussions and information exchanges with ICOT researchers.

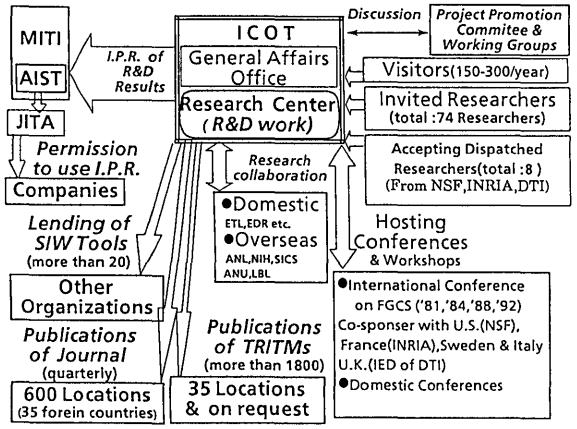


Figure 4-1 R&D result distribution and research collaboration

We could, for example, produce GHC, a core language of the parallel system, by discussion with researchers working on Parlog and Concurrent Prolog. We could, also, improve the performance of the PSI system by introducing the WAM instruction set proposed by Professor Warren.

We have several programs for distributing the R&D results of this project, to exchange information and to collaborate with researchers and organizations.

① One important way to present R&D activities and results is publication and distribution of ICOT journals and technical papers. We have published and distributed quarterly journals, which contain introductions of ICOT activities, and technical papers to more than 600 locations in 35 countries.

We have periodically published and sent more than 1800 technical papers to around 30 overseas locations. We have sent TRs (Technical Reports) and TMs (Technical Memos) on request to foreign addresses. These technical papers consist of more than 700 TRs and 1100 TMs published since the beginning of this project up to January 1992. A third of these technical papers are written in English.

② In the second program ICOT researchers discuss research matters and exchange information with outside researchers.

- ICOT researchers have made more than 450 presentations at international conferences and workshops, and at around 1800 domestic conferences and workshops. They have visited many foreign research organizations to discuss specific research themes and to explain ICOT activities.
- Every year, we have welcomed around 150 to 300 foreign researchers and specialists in other fields to exchange information with them and explain ICOT activities to them.
- As already described in the previous chapter, we have so far invited 74 active researchers from specific technical fields related to FGCS technologies. We have also received six long-term visiting researchers dispatched from foreign governmental organization based on agreement. These visiting researchers conducted research at ICOT and published the results of that research.

③ We sponsored the following symposiums and workshops to disseminate and exchange information on the R&D results and on ICOT activities.

- We hosted the International Conference on FGCS'84 in November 1984. Around 1,100 persons participated and the R&D results of the initial stage were presented. This followed the International Conference on FGCS'81, in which the FGCS project plan was presented. We also hosted the International Conference on FGCS'88 in November 1988. 1,600 persons participated in this symposium, and we presented the R&D results of the intermediate stage.
- We have held
  7 Japan-Sweden (or Japan-Sweden-Italy) workshops since 1983 (co-sponsored with institute or universities in Sweden and Italy),
  4 Japan-France AI symposiums since 1986, (co-sponsored with INRIA of France),
  4 Japan-U.S. AI symposiums since 1987 (co-sponsored with NSF of U.S.A.), and
  2 Japan-U.K. workshops since 1989 (co-sponsored with DTI of U.K.).

  Participating researchers have become to known each other well through presentations and discussions during these symposiums and workshops.
- We have also hosted domestic symposiums on this project and logic programming conferences every year.

④ Because the entire R&D cost of this project has been provided by the government, such intellectual property rights (IPR) as patents, which are produced in this project, belong to the Japanese government. These IPR are managed by AIST (Agency of Industrial Science and Technology). Any company wishing to produce commercial products that use any of these IPR must get permission to use them from AIST. For example, PSI and SIMPOS have already been commercialized by companies licensed by AIST. The framework for managing IPR must

impartially utilize IPR acquired through this project. That is, impartial permission to domestic and foreign companies, and among participating companies or others is possible because of AIST.

⑤ Software tools developed in this project that are not yet managed as IPR by AIST can be used by other organizations for non-commercial aims. These software tools are distributed by ICOT according to the research tools permission procedure. We, now, have more than 20 software tools, such as PIMOS, PDSS, Kappa-II, the A'um system, LTB, the CAP system, the cu-prolog system and the TRS generator.
In other cases, we make the source codes of some programs public by printing them in technical papers.

⑥ On specific research themes in the logic programming field, we have collaborated with organizations such as Argonne National Laboratory (ANL), National Institute of Health (NIH), Lawrence Berkeley Laboratory (LBL), Swedish Institute of Computer Science (SICS) and Australia National University (ANU).

5 Forecast of Some Aspects of 5G Machines

LSI technologies have advance in accordance with past trends. Roughly speaking, the memory capacity and the number of gates of a single chip quadruple every three years. The number of boards for the CPU of an inference machine was more than ten for PSI-Ⅰ, but only three for PSI-Ⅱ and single board for PIM.

The number of boards for 80M bytes memory was 16 for PSI-Ⅰ, but only four for PSI-Ⅱ and a single for PIM (m).

Figure 5-1 shows the anticipated trend in board numbers for one PE (processor element: CPU and memory) and cost for one PE based on the actual value of inference machines developed by this project.

The trend shows that, by the year 2000, around ten PEs will fit on one board, around 100 PEs will fit in one desk side cabinet, and 500 to a 1,000 PEs will fit in a large cabinet. This trend also shows that the cost of one PE will halve every three years.

Figure 5-2 shows the performance trends of 5G machines based on the actual performance of inference machines developed by this project.

The sequential inference processing performance for one PE quadrupled every three years. The improvement in parallel inference processing performance for one PE was not as large as it was for sequential processing, because PIM performance is estimated at around two and one half times that

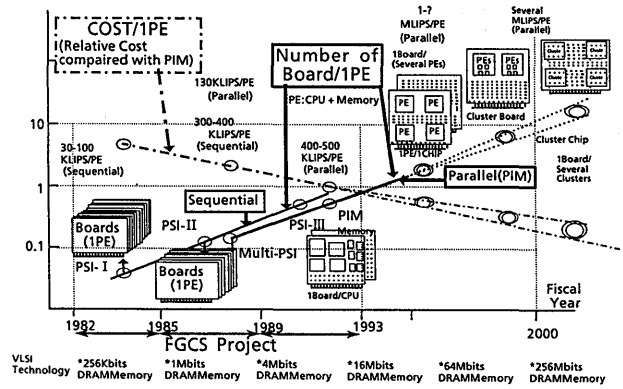


Figure 5-1 Size and cost trends of 5G machines

of multi-PSI. Furthermore, Figure 5-2 shows the performance of one board for both sequential and parallel processing, and the performance of a conventional micro-processor with CISC and RISC technology. In this figure, future improvements in the performance of one PE are estimated to be rather lower than a linear extension of past values would indicate because of the uncertainty of whether future technology will be able to elicit such performance improvements. Performance for one board is estimated at about 20 MLIPS, which is 100 times faster than PIM. Thus, a parallel machine with a large cabinet size could have 1 GLIPS. These parallel systems will have the processing speeds needed for various knowledge processing applications in the near future.

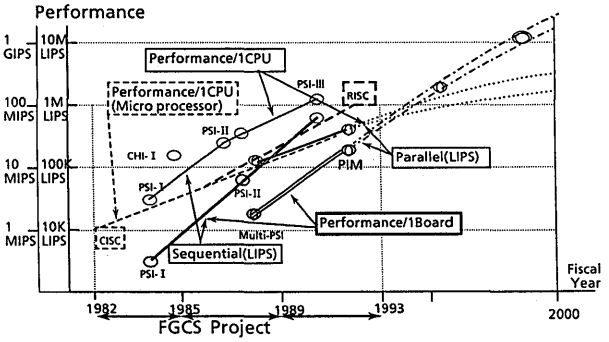


Figure 5-2 Performance trends of 5G machines

Several parallel applications in this project, such as CAD, theorem provers, genetic information processing, natural language processing, and legal reasoning are described in Chapter 2. These applications are distributed in various fields and aim at cultivating new parallel processing application fields.

We believe that parallel machine applications will be extended to various areas in industry and society, because parallel technology will become

common for computers in the near future. Parallel application fields will expand gradually according to function expansion by the use of advanced parallel processing and knowledge processing technologies.

6 Final Remarks

I believe that we have shown the basic framework of the fifth generation computer based on logic programming to be more than mere hypothesis. By the end of the initial stage, we had shown the fifth generation computer to be viable and efficient through the development of PSI, SIMPOS and various experimental software systems written in ESP and Prolog.

I believe that by the end of the intermediate stage, we had shown the possibility of realizing the fifth generation computer through the development of a parallel logic programming software environment which consisted of multi-PSI and PIMOS.

And I hope you can see the possibility of an era of parallel processing arriving in the near future by looking at the prototype system and the R&D results of the FGCS Project.

Acknowledgment

This project has been carried out through the efforts of the researchers at ICOT, and with the support of MITI and many others outside of ICOT. We wish to extend our appreciation to them all for the direct and indirect assistance and co-operation they have provided.

References

[Motooka, et al 1981] Proceedings of the International Conference on Fifth Generation Computer Systems, 1981, JIPDEC

[Kawanobe, et al 1984] K.Kawanobe, et al. ICOT Research and Development, Proceeding of the International Conference on Fifth Generation Computer Systems 1984, 1984, ICOT

[Kurozumi, et al 1987] T.Kurozumi, et al. Fifth Generation Computer Systems Project, 1987, ICOT TM303

[Kurozumi, et al 1988] T.Kurozumi, et al. ICOT Research and development, Proceedings of the International Conference on Fifth Generation Computer Systems 1988, 1988, ICOT

[Kurozumi, 1990] T.Kurozumi. Fifth Generation Computer Systems Project-Outline of Plan and Results, 1990, ICOT TM-996

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Summary of Basic Research Activities of the FGCS Project

Koichi Furukawa

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
furukawa@icot.or.jp

Abstract

The Fifth Generation Computer Project was launched in 1982, with the aim of developing parallel computers dedicated to knowledge information processing. It is commonly believed to be very difficult to parallelize knowledge processing based on symbolic computation. We conjectured that *logic programming* technology would solve this difficulty.

We conducted our project while stressing two seemingly different aspects of logic programming: one was establishment of a new information technology, and the other was pursuit of basic AI and software engineering research.

In the former, we developed a concurrent logic programming language, GHC, and its extension for practical parallel programming, KL1. The invention of GHC/KL1 enabled us to conduct parallel research on the development of software technology and parallel hardware dedicated to the new language.

We also developed several constraint logic programming languages which are very promising as high level languages for AI applications. Though most of them are based on sequential Prolog technology, we are now integrating constraint logic programming and concurrent logic programming and developing an integrated language, GDCC.

In the latter, we investigated many fundamental AI and software engineering problems including hypothetical reasoning, analogical inference, knowledge representation, theorem proving, partial evaluation and program transformation.

As a result, we succeeded in showing that logic programming provides a very firm foundation for many aspects of information processing: from advanced software technology for AI and software engineering, through system programming and parallel programming, to parallel architecture.

The research activities are continuing and latest as well as earlier results strongly indicate the truth of our conjecture and also the fact that our approach is appropriate.

1 Introduction

In the Fifth Generation Computer Project, two main research targets were pursued: knowledge information processing and parallel processing. Logic programming was adopted as a key technology for achieving both targets simultaneously. At the beginning of the project, we adopted Prolog as our vehicle to promote the entire research of the project. Since there were no systematic research attempts based on Prolog before our project, there were many things to do, including the development of a suitable workstation for the research, experimental studies for developing a knowledge-based system in Prolog and investigation into possible parallel architecture for the language. We rapidly succeeded in promoting research in many directions.

From this research, three achievements are worth noting. The first is the development of our own workstation dedicated to ESP, Extended Self-contained Prolog. We developed an operating system for the workstation completely in ESP [Chikayama 88]. The second is the application of partial evaluation to meta programming. This enabled us to develop a compiler for a new programming language by simply describing an interpreter of the language and then partially evaluating it. We applied this technique to derive a bottom-up parser for context free grammar given a bottom up interpreter for them. In other words, partial evaluation made meta programming useful in real applications. The third achievement was the development of constraint logic programming languages. We developed two constraint logic programming languages: CIL and CAL. CIL is for natural language processing and is based on the incomplete data structure for representing "Complex Indeterminates" in situation theory. It has the capability to represent structured data like Minsky's frame and any relationship between slots' values can be expressed using constraints. CIL was used to develop a natural language understanding system called DUALS. Another constraint logic programming language, CAL, is for non-linear equations. Its inference is done using the Buchberger algorithm for computing the Gröbner Basis which is a variant of the Knuth-Bendix completion algorithm for a term rewriting

system.

We encountered one serious problem inherent to Prolog: that was the lack of concurrency in the fundamental framework of Prolog. We recognized the importance of concurrency in developing parallel processing technologies, and we began searching for alternative logic programming languages with the notion of concurrency.

We noticed the work by Keith Clark and Steve Gregory on Relational Language [ClarkGregory 81] and Ehud Shapiro on Concurrent Prolog [Shapiro 83]. These languages have a common feature of committed choice nondeterminism to introduce concurrency. We devoted our efforts to investigating these languages carefully and Ueda finally designed a new committed choice logic programming language called GHC [Ueda 86a] [UedaChikayama 90], which has simpler syntax than the above two languages and still have similar expressiveness. We recognized the importance of GHC and adopted it as the core of our kernel language, KL1, in this project. The introduction of KL1 made it possible to divide the entire research project into two parts: the development of parallel hardware dedicated to KL1 and the development of software technology for the language. In this respect, the invention of GHC is the most important achievement for the success of the Fifth Generation Computer Systems project.

Besides these language oriented researches, we performed many fundamental researches in the field of artificial intelligence and software engineering based on logic and logic programming. They include researches on nonmonotonic reasoning, hypothetical reasoning, abduction, induction, knowledge representation, theorem proving, partial evaluation and program transformation. We expected that these researches would become important application fields for our parallel machines by the affinity of these problems to logic programming and logic based parallel processing. This is now happening.

In this article, we first describe our research efforts in concurrent logic programming and in constraint logic programming. Then, we discuss our recent research activities in the field of software engineering and artificial intelligence. Finally, we conclude the paper by stating the dirction of future research.

2 Concurrent Logic Programming

In this section, we pick up two important topics in concurrent logic programming research in the project. One is the design principles of our concurrent logic programming language Flat GHC (FGHC) [Ueda 86a] [UedaChikayama 90], on which the aspects of KL1 as a concurrent language is based. The other is search paradigms in FGHC. As discussed later, one drawback of FGHC, viewing as a logic programming language, is the lack of search capability inherent in Prolog. Since the capability is related to the notion of completeness in logic programming, recovery of the ability is essential.

2.1 Design Principles of FGHC

The most important feature of FGHC is that there is only one syntactic extension to Prolog, called the commitment operator and represented by a vertical bar "|". A commitment operator divides an entire clause into two parts called the guard part (the left-hand side of the bar) and the body part (the right-hand side). The guard of a clause has two important roles: one is to specify a condition for the clause to be selected for the succeeding computation, and the other is to specify the synchronization condition. The general rule of synchronization in FGHC is expressed as *dataflow synchronization*. This means that computation is suspended until sufficient data for the computation arrives. In the case of FGHC, guard computation is suspended until the caller is sufficiently instantiated to judge the guard condition. For example, consider how a ticket vending machine works. After receiving money, it has to wait until the user pushes a button for the destination. This waiting is described as a clause such that "if the user pushed the 160-yen button, then issue a 160-yen ticket".

The important thing is that dataflow synchronization can be realized by a simple rule governing head unification which occurs when a goal is executed and a corresponding FGHC clause is called: the information flow of head unification must be one way, from the caller to the callee. For example, consider a predicate representing service at a front desk. Two clauses define the predicate: one is for during the day, when more customers are expected, and another is for after-hours, when no more customers are expected. The clauses have such definitions as:

```
serve([First | Rest]) :- <extra-condition> |
     do_service(First), serve(Rest).
serve([]) :- true | true.
```

Besides the *serve* process, there should be another process *queue* which makes a waiting queue for service. The top level goal looks like:

```
?- queue(Xs), serve(Xs).
```

where "?-" is a prompt to the user at the terminal. Note that the execution of this goal generates two processes, queue and serve, which share a variable Xs. This shared variable acts as a channel for data transfer from one process to the other. In the above example, we assume that the queue process instantiates Xs and the serve process reads the value. In other words, queue acts as a generator of the value of Xs and serve acts as the consumer. The process queue instantiates Xs either to a

list of servees represented by [<first-servee>, <second-servee>,...] or to an empty list []. Before the instantiation, the value of Xs remains undefined.

Suppose Xs is undefined. Then, the head unification invoked by the goal serve(Xs) suspends because the equations Xs = [First | Rest] and Xs = [] cannot be solved without instantiating Xs. But such instantiation violates the rule of one-way unification. Note that the term [First | Rest] in the head of serve means that the clause expects a non-empty list to be given as the value of the argument. Similarly, the term [] expects an empty list to be given. Now, it is clear that the unidirectionality of information flow realizes dataflow synchronization.

This principle is very important in two aspects: one is that the language provides a natural tool for expressing concurrency, and the other is that the synchronization mechanism is simple enough to realize very efficient parallel implementation.

2.2 Search Paradigms in FGHC

There is one serious drawback to FGHC because of the very nature of committed choice; that is, it no longer has an automatic search capability, which is one of the most important features of Prolog. Prolog achieves its search capability by means of automatic backtracking. However, since committed choice uniquely determines a clause for succeeding computation of a goal, there is no way of searching for alternative branches other than the branch selected. The search capability is related to the notion of completeness of the logic programming computation procedure and the lack of the capability is very serious in that respect.

One could imagine a seemingly trivial way of realizing search capability by means of *or-parallel* search: that is, to copy the current computational environment which provides the binding information of all variables that have appeared so far and to continue computations for each alternative case in parallel. But this does not work because copying non-ground terms is impossible in FGHC. The reason why it is impossible is that FGHC cannot guarantee when actual binding will occur and there may be a moment when a variable observed at some processor remains unchanged even after some goal has instantiated it at a different processor.

One might ask why we did not adopt a Prolog-like language as our kernel language for parallel computation. There are two main reasons. One is that, as stated above, Prolog does not have enough expressiveness for concurrency, which we see as a key feature not only for expressing concurrent algorithms but also for providing a framework for the control of physical parallelism. The other is that the execution mechanism of Prolog-like languages with a search capability seemed too complicated to develop efficient parallel implementations.

We tried to recover the search capability by devising programming techniques while keeping the programming language as simple as possible. We succeeded in inventing several programming methods for computing all solutions of a problem which effectively achieve the completeness of logic programming. Three of them are listed as follows:

(1) Continuation-based method [Ueda 86b]
(2) Layered stream method [OkumuraMatsumoto 87]
(3) Query compilation method [Furukawa 92]

In this paper, we pick up (1) and (3), which are complementary to each other. The continuation-based method is suitable for the efficient processing of rather algorithmic problems. An example is to compute all ways of partitioning a given list into two sublists by using *append*. This method mimics the computation of *OR-parallel* Prolog using *AND-parallelism* of FGHC. *AND-serial* computation in Prolog is translated to *continuation* processing which remembers continuation points in a stack. The intermediate results of computation are passed from the preceding goals to the next goals through the continuation stack kept as one of the arguments of the FGHC goals. This method requires input/output mode analysis before translating a Prolog program into FGHC. This requirement makes the method impractical for database applications because there are too many possible input-output modes for each predicate.

The query compilation method solves this problem. This method was first introduced by Fuchi [Fuchi 90] when he developed a bottom-up theorem prover in KL1. In his coding technique, the multiple binding problem is avoided by reversing the role of the caller and the callee in straightforward implementation of database query evaluation. Instead of trying to find a record (represented by a clause) which matches a given query pattern represented by a goal, his method represents each query component with a compiled clause, represents a databasae with a data structure passed around by goals, and tries to find a query component clause which matches a goal representing a record and recurses the process for all potentially applicable records in the database[1]. Since every record is a ground term, there is no variable in the caller. Variable instantiation occurs when query component clauses are searched and an appropriate clause representing a query component is found to match a currently processed record. Note that, as a result of reversing the representation of queries and databases from straightforward representation, the information flow is now from the caller (database) to the callee (a query component). This inversion of information flow avoids deadlock in query processing. Another important trick is that each time a query clause is called, a fresh variable is created for each variable in the query component. This mechanism is used for making a new environment

[1] We need an *auxiliary* query clause which matches every record after failing to match the record to all the *real* query clauses.

for each OR-parallel computation branch. These tricks make it possible to use KL1 variables to represent object level variables in database queries and, therefore, we can avoid different compilation of the entire database and queries for each input/output mode of queries.

The new coding method stated above is very general and there are many applications which can be programmed in this way. The only limitation of this approach is that the database must be more instantiated than queries. In bottom-up theorem proving, this requirement is referred to as the range-restrictedness of each axiom. Range-restrictedness means that, after successfully finding ground model elements satisfying the antecedent of an axiom, the new model element appearing as the consequent of the axiom must be ground.

This restriction seems very strong. Indeed, there are problems in the theorem proving area which do not satisfy the condition. We need a top-down theorem prover for such problems. However, many real life problems satisfy the range-restrictedness because they almost always have finite concrete models. Such problems include VLSI-CAD, circuit diagnosis, planning, and scheduling. We are developing a parallel bottom-up theorem prover called MGTP (Model Generation Theorem Prover) [FujitaHasegawa 91] based on SATCHMO developed by Manthey and Bry [MantheyBry 88]. We are investigating new applications to utilize the theorem prover. We will give an example of computing abduction using MGTP in Section 5.

3 Constraint Logic Programming

We began our constraint logic programming research almost at the beginning of our project, in relation to the research on natural language processing. Mukai [MukaiYasukawa 85] developed a language called CIL (Complex Indeterminates Language) for the purpose of developing a computational model of situation semantics. A *complex indeterminate* is a data structure allowing partially specified terms with indefinite arity. During the design phase of the language, he encountered the idea of *freeze* in Prolog II by Colmerauer [Colmerauer 86]. He adopted *freeze* as a proper control structure for our CIL language.

From the viewpoint of constraint satisfaction, CIL only has a passive way of solving constraint, which means that there is no active computation for solving constraints such as constraint propagation or solving simultaneous equations. Later, we began our research on constraint logic programming involving active constraint solving. The language we developed is called CAL. It deals with non-linear equations as expressions to specify constraints. Three events triggered the research: one was our preceding efforts on developing a term rewriting system called METIS for a theorem prover of linear algebra [OhsugaSakai 91]. Another event was our encounter with Buchberger's algorithm for computing the Gröbner Basis for solving non-linear equations. Since the algorithm is a variant of the Knuth-Bendix completion algorithm for a term rewriting system, we were able to develop the system easily from our experience of developing METIS. The third event was the development of the CLP(X) theory by Jaffar and Lassez which provides a framework for constraint logic programming languages [JaffarLassez 86].

There is further remarkable research on constraint logic programming in the field of general symbol processing [Tsuda 92]. Tsuda developed a language called cu-Prolog. In cu-Prolog, constraints are solved by means of program transformation techniques called unfold/fold transformation (these will be discussed in more detail later in this paper, as an optimization technique in relation to software engineering). The unfold/fold program transformation is used here as a basic operation for solving combinatorial constraints among terms. Each time the transformation is performed, the program is modified to a syntactically less constrained program. Note that this basic operation is similar to *term rewriting*, a basic operation in CAL. Both of these operations try to rewrite programs to get certain canonical forms. The idea of cu-Prolog was introduced by Hasida during his work on *dependency propagation* and *dynamical programming* [Hasida 92]. They succeeded in showing that context-free parsing, which is as efficient as *chart parsing*, emerges as a result of dependency propagation during the execution of a program given as a set of grammar rules in cu-Prolog. Actually, there is no need to construct a parser. cu-Prolog itself works as an efficient parser.

Hasida [Hasida 92] has been working on a fundamental issue of artificial intelligence and cognitive science from the aspect of a computational model. In his computation model of dynamical programming, computation is controlled by various kinds of *potential energies* associated with each atomic constraint, clause, and unification. Potential energy reflects the degree of constraint violation and, therefore, the reduction of energy contributes constraint resolution.

Constraint logic programming greatly enriched the expressiveness of Prolog and is now providing a very promising programming environment for applications by extending the domain of Prolog to cover most AI problems.

One big issue in our project is how to integrate constraint logic programming with concurrent logic programming to obtain both expressiveness and efficiency.

This integration, however, is not easy to achieve because (1) constraint logic programming focuses on a control scheme for efficient execution specific to each constraint solving scheme, and (2) constraint logic programming essentially includes a search paradigm which re-

quires some suitable support mechanism such as automatic backtracking.

It turns out that the first problem can be processed efficiently, to some extent, in the concurrent logic programming scheme utilizing the data flow control method. We developed an experimental concurrent constraint logic programming language called GDCC (Guarded Definite Clauses with Constraints), implemented in KL1 [HawleyAiba 91]. GDCC is based on an ask-tell mechanism proposed by Maher [Maher 87], and extended by Saraswat [Saraswat 89]. It extends the guard computation mechanism from a simple one-way unification solving problem to a more general provability check of conditions in the guard part under a given set of constraints using the *ask* operation. For the body computation, constraint literals appearing in the body part are added to the constraint set using the *tell* operation. If the guard conditions are not known to be provable because of a lack of information in the constraints set, then computation is suspended. If the conditions are disproved under the constraints set, then the guard computation fails. Note that the provability check controls the order of constraint solving execution. New constraints appearing in the body of a clause are not included in the constraint set until the guard conditions are known to be provable.

The second problem of realizing a search paradigm in a concurrent constraint logic programming framework has not been solved so far. One obvious way is to develop an OR-parallel search mechanism which uses a full unification engine implemented using ground term representation of logical variables [Koshimura *et al.* 91]. However, the performance of the unifier is 10 to 100 times slower than the built in unifier and, as such, it is not very practical. Another possible solution is to adopt the new coding technique introduced in the previous section. We expect to be able to efficiently introduce the search paradigm by applying the coding method. The paradigm is crucial if parallel inference machines are to be made useful for the numerous applications which require high levels of both expressive and computational power.

4 Advanced Software Engineering

Software engineering aims at supporting software development in various dimensions; increase of software productivity, development of high quality software, pursuit of easily maintainable software and so on. Logic programming has great potential for many dimensions in software engineering. One obvious advantage of logic programming is the affinity for correctness proof when given specifications. Automatic debugging is a related issue. Also, there is a high possibility of achieving automatic program synthesis from specifications by applying proof techniques as well as from examples by applying induction. Program optimization is another promising direction where logic programming works very well.

In this section, two topics are picked up: (1) meta programming and its optimization by partial evaluation, and (2) unfold/fold program transformation.

4.1 Meta Programming and Partial Evaluation

We investigated meta programming technology as a vehicle for developing knowledge-based systems in a logic programming framework inspired by Bowen and Kowalski's work [BowenKowalski 83]. It was a rather direct way to realize a knowledge assimilation system using the meta programming technique by regarding integrity constraints as meta rules which must be satisfied by a knowledge base. One big problem of the approach was its inefficiency due to the meta interpretation overhead of each object level program. We challenged the problem and Takeuchi and Furukawa [TakeuchiFurukawa 86] made a breakthrough in the problem by applying the optimization technique of partial evaluation to meta programs. We first derived an efficient *compiled* program for an expert system with uncertainty computation given a meta interpreter of rules with certainty factor. In this program, we succeeded in getting three times speedup over the original program. Then, we tried a more non-trivial problem of developing a meta interpreter of a bottom-up parser and deriving an efficient *compiled* program given the interpreter and a set of grammar rules. We succeeded in obtaining an object program known as BUP, developed by Matsumoto [Matsumoto *et al.* 83]. The importance of the BUP meta-interpreter is that it is not a vanilla meta-interpreter, an obvious extension of the Prolog interpreter in Prolog, because the control structure is totally different from Prolog's top-down control structure.

After our first success of applying partial evaluation techniques in meta programming, we began the development of a self-applicable partial evaluator. Fujita and Furukawa [FujitaFurukawa 88] succeeded in developing a simple self-applicable partial evaluator. We showed that the partial evaluator itself was a meta interpreter very similar to the following Prolog interpreter in Prolog:

```
solve(true).
solve((A,B)) :-  solve(A),   solve(B).
solve(A)     :- clause(A,B), solve(B).
```

where it is assumed that for each program clause, H :- B, a unit clause, `clause(`H`,`B`)`, is asserted[2]. A goal, `solve(`G`)`, simulates an immediate execution of the subject goal, G, and obtains the same result.

This simple definition of a Prolog self-interpreter, `solve`, suggests the following *partial solver,* `psolve`.

[2] `clause(_,_)` is available as a built-in procedure in the DECsystem-10 Prolog system.

```
psolve(true,true).
psolve((A,B),(RA,RB)) :-
     psolve(A,RA), psolve(B,RB).
psolve(A,R) :-
     clause(A,B), psolve(B,R).
psolve(A,A) :- '$suspend'(A).
```

The partial solver, psolve(G,R), partially solves a given goal, G, returning the result, R. The result, R, is called the *residual goal(s)* for the given goal, G. The residual goal may be true when the given goal is totally solved, otherwise it will be a conjunction of subgoals, each of which is a goal, R_i, suspended to be solved at '$suspend'($R_i$), for some reason. An *auxiliary predicate*, '$suspend'($P$), is defined for each goal pattern, P, by the user.

Note that psolve is related to solve as:

```
solve(G) :- psolve(G,R), solve(R).
```

That is, a goal, G, succeeds if it is partially solved with the residual goal, R, and R in turn succeeds (is totally solved). The total solution for G is thus split into two tasks: partial solution for G and total solution for the residual goal, R.

We developed a self-applicable partial evaluator by modifying the above psolve program. The main modification is the treatment of built-in predicates in Prolog and those predicates used to define the partial evaluator itself to make it self-applicable. We succeeded in applying the partial evaluator to itself and generated a compiler by partially evaluating the psolve program with respect to a given interpreter, using the identical psolve. We further succeeded in obtaining a compiler generator, which generates different compilers given different interpreters, by partially evaluating the psolve program with respect to itself, using itself.

Theoretically, it was known that self-application of a partial evaluator generates compilers and a compiler generator [Futamura 71]. There were many attempts to realize self-applicable partial evaluators in the framework of functional languages for a long time, but no successes were reported until very recently [Jones *et al.* 85], [Jones *et al.* 88], [GomardJones 89]. On the other hand, we succeeded in developing a self-applicable partial evaluator in a Prolog framework in a very short time and also in a very elegant way. This proves some merits of logic programming languages over functional programming languages, especially in its binding scheme based on unification.

4.2 Unfold/Fold Program Transformation

Program transformation provides a powerful methodology for the development of software, especially the derivation of efficient programs either from their formal specification or from decralative but possibly inefficient programs. Programs written in declarative form are often inefficient under Prolog's standard left to right control rule. Typical examples are found in programs based on a generate and test paradigm. Seki and Furukawa [SekiFurukawa 87] developed a program transformation method based on unfolding and folding for such programs. We will explain the idea in some detail. Let gen_test(L) be a predicate defined as follows:

```
gen_test(L) :- gen(L), test(L).
```

where L is a variable representing a list, gen(L) is a generator of the list L, and test(L) is a tester for L. Assume both gen and test are incremental and are defined as follows:

```
gen([]).
gen([X|L]) :- gen_element(X), gen(L).

test([]).
test([X|L]) :- test_element(X), test(L).
```

Then, it is possible to fuse two processes gen and test by applying unfold/fold transformation as follows:

```
gen_test([X|L]) :- gen([X|L]), test([X|L]).
```

unfold at gen and test

```
gen_test([X|L]) :- gen_element(X), gen(L),
    test_element(X), test(L).
```

fold by gen_test

```
gen_test([X|L]) :- gen_element(X),
    test_element(X), gen_test(L).
```

If the tester is not incremental, the above unfold/fold transformation does not work. One example is to test that all elements in the list are different from each other. In this case, the test predicate is defined as follows:

```
test([]).
test([X|L]) :- non_member(X,L), test(L).

non_member(_,[]).
non_member(X,[Y|L]):-
    dif(X,Y), non_member(X,L).
```

where dif(X,Y) is a predicate judging that X is not equal to Y. Note that this test predicate is not incremental because a test for the first element X of the list requires the information of the entire list. The solution we gave to this problem was to replace the test predicate with an equivalent predicate with incrementality. Such an equivalent program test' is obtained by adding an *accumulator* as an extra argument of the test predicate defined as follows:

```
test'([],_).
test'([X|L],Acc) :-
    non_member(X,Acc), test'(L,[X|Acc]).
```

The relationship between test and test' is given by the following theorem:

Theorem

$$\texttt{test(L)} \equiv \texttt{test'(L,[])}$$

Now, the original gen_test program becomes

```
gen_test(L) :- gen(L), test'(L,[]).
```

We need to introduce the following new predicate to perform the unfold/fold transformation:

```
gen_test'(L,Acc) :- gen(L), test'(L,Acc).
```

By applying a similar transformation process as before, we get the following fused recursive program of gen_test':

```
gen_test'([],_).
gen_test'([X|L],Acc) :- gen_element(X),
    non_member(X,Acc), gen_test'(L,[X|Acc]).
```

By symbolically computing the two goals

```
?- test([X1,...,Xn]).

?- test'([X1,...,Xn]).
```

and comparing the results, one can find that the reordering of pair-wise comparisons by the introduction of the accumulator is analogous to the exchange of double summation $\Sigma_{i=1}^{N}\Sigma_{j=i}^{N}x_{ij} = \Sigma_{j=1}^{N}\Sigma_{i=1}^{j}x_{ij}$. Therefore, we refer to this property as structural commutativity.

One of the key problems of unfold/fold transformation is the introduction of a new predicate such as gen_test' in the last example. Kawamura [Kawamura 91] developed a syntactic rule for finding suitable new predicates. There were several attempts to find appropriate new predicates using domain dependent heuristic knowledge, such as append optimization by the introduction of difference list representation. Kawamura's work provides some general criteria for selecting candidates for new predicates. His method first analyzes a given program to be transformed and makes a list of patterns which may possibly appear in the definition of new predicates. This can be done by unfolding a given program and properly generalizing all resulting patterns to represent them with a finite number of distinct patterns while avoiding over-generalization. One obvious strategy to avoid over-generalization is to perform least general generalization by Plotkin [Plotkin 70]. Kawamura also introduced another strategy for suppressing unnecessary generalization: a subset of clauses of which the head can be unifiable to each pattern is associated with the pattern and only those patterns having the same associated subset of clauses are generalized. Note that a goal pattern is unfolded only by clauses belonging to the associated subset. Therefore the suppression of over-generalization also suppresses unnecessary expansion of clauses by unnecessary unfolding.

5 Logic-based AI Research

For a long time, deduction has played a central role in research on logic and logic programming. Recently, two other inferences, abduction and induction, received much attention and much research has been done in these new directions. These directions are related to fundamental AI problems that are open-ended by their nature. They include the frame problem, machine learning, distributed problem solving, natural language understanding, common sense reasoning, hypothetical reasoning and analogical reasoning. These problems require non-deductive inference capabilities in order to solve them.

Historically, most AI research on these problems adopted ad hoc heuristic methods reflecting problem structures. There was a tendency to avoid a logic based formal approach because of a common belief in the limitation of the formalism. However, the limitation of logical formalism comes only from the deductive aspect of logic. Recently it has been widely recognized that abduction and induction based on logic provide a suitable framework for such problems requiring open-endedness in their formalism. There is much evidence to support this observation.

- In natural language understanding, unification grammar is playing an important role in integrating syntax, semantics, and discourse understanding.
- In non-monotonic reasoning, logical formalism such as circumscription and default reasoning and its compilation to logic based programs are studied extensively.
- In machine learning, there are many results based on logical frameworks such as the Model Inference System, inverse resolution, and least general generalization.
- In analogical reasoning, analogy is naturally described in terms of a formal inference rule similar to logical inference. The associated inference is deeply related to abductive inference.

In the following, three topics related to these issues are picked up: they are hypothetical reasoning, analogy, and knowledge representation.

5.1 Hypothetical Reasoning

A logical framework of hypothetical reasoning was studied by Poole et al. [Poole *et al.* 87]. They discussed the relationship among hypothetical reasoning, default reasoning and circumscription, and argued that hypothetical reasoning is all that is needed because it is simply and efficiently implemented and is powerful enough to implement other forms of reasoning. Recently, the relationship of these formalisms was studied in more detail and many attempts were made to translate non-monotonic reasoning problems into equivalent logic programs with negation as failure.

Another direction of research was the formulation of abduction and its relationship to negation as failure. There was also a study of the model theory of a class of logic programs, called general logic programs, allowing negation by failure in the definition of bodies in the clausal form. By replacing negation-by-failure predicates by corresponding abducible predicates which usually give negative information, we can formalize negation by failure in terms of abduction [EshghiKowalski 89]

A proper semantics of general logic programs is given by stable model semantics [GelfondLifschitz 88]. It is a natural extension of least fixpoint semantics. The difference is that there is no T_P operator to compute the stable model directly, because we need a complete model for checking the truth value of the literal of negation by failure in bottom-up fixpoint computation. Therefore, we need to refer to the model in the definition of the model. This introduces great difficulty in computing stable models. The trivial way is to assume all possible models and see whether the initial models are the least ones satisfying the programs or not. This algorithm needs to search for all possible subsets of atoms to be generated by the programs and is not realistic at all.

Inoue [Inoue *et al.* 92] developed a much more efficient algorithm for computing all stable models of general logic programs. Their algorithm is based on bottom-up model generation method. Negation-by-failure literals are used to introduce hypothetical models: ones which assume the truth of the literals and the others which assume that they are false. To express assumed literals, they introduce a modal operator. More precisely, they translate each rule of the form:

$$A_l \leftarrow A_{l+1} \wedge \ldots \wedge A_m, \mathit{not}\, A_{m+1} \wedge \ldots \wedge \mathit{not}\, A_n$$

to the following disjunctive clause which does not contain any negation-by-failure literals:

$$A_{l+1} \wedge \ldots \wedge A_m \rightarrow$$
$$(\mathsf{NK}A_{m+1} \wedge \ldots \wedge \mathsf{NK}A_n \wedge A_l) \vee \mathsf{K}A_{m+1} \vee \ldots \vee \mathsf{K}A_n .$$

The reason why we express the clause with the antecedent on the left hand side is that we intend to use this clause in a bottom-up way; that is, from left to right. In this expression, $\mathsf{NK}A$ means that we *assume* that A is false, whereas, $\mathsf{K}A$ means that we *assume* that A is true. Although K and NK are modal operators, we can treat $\mathsf{K}A$ and $\mathsf{NK}A$ as new predicates independent from A by adding the following constraints:

$$\mathsf{NK}A,\ A \rightarrow \quad \text{for every atom } A. \tag{1}$$

$$\mathsf{NK}A,\ \mathsf{K}A \rightarrow \quad \text{for every atom } A. \tag{2}$$

By this translation, we obtain a set of clauses in first order logic and therefore it is possible to compute all possible models for the set using a first order bottom-up theorem prover, MGTP, described in Section 2. After computing all possible models for the set of clauses, we need to select only those models M which satisfy the following condition:

$$\text{For every ground atom } A, \text{ if } \mathsf{K}A \in M, \text{ then } A \in M. \tag{3}$$

Note that this translation scheme defines a coding method of original general logic programs which may contain negation by failure in terms of pure first order logic. Note also that the same technique can be applied in computing abduction, which means to find possible sets of hypotheses explaining the observation and not contradicting given integrity constraints.

Satoh and Iwayama [SatohIwayama 92] independently developed a top-down procedure for answering queries to a general logic program with integrity constraints. They modified an algorithm proposed by Eshghi and Kowalski [EshghiKowalski 89] to correctly handle situations where some proposition must hold in a model, like the requirement of (3).

Iwayama and Satoh [IwayamaSatoh 91] developed a mixed strategy combining bottom-up and top-down strategies for computing the stable models of general logic programs with constraints. The procedure is basically bottom-up. The top-down computation is related to the requirement of (3) and as soon as a hypothesis of $\mathsf{K}A$ is asserted in some model, it tries to prove A by a top-down expectation procedure.

The formalization of abductive reasoning has a wide range of applications including computer aided design and fault diagnosis. Our approach provides a uniform scheme for representing such problems and solving them. It also provides a way of utilizing our parallel inference machine, PIM, for solving these complex AI problems.

5.2 Formal Approach to Analogy

Analogy is an important reasoning method in human problem solving. Analogy is very helpful for solving problems which are very difficult to solve by themselves. Analogy guides the problem solving activities using the knowledge of how to solve a similar problem. Another aspect of analogy is to extract good guesses even when there is not enough information to explain the answer.

There are three major problems to be solved in order to mechanize analogical reasoning [Arima 92]:

- searching for an appropriate base of analogy with respect to a given target,
- selecting important properties shared by a base and a target, and
- selecting properties to be projected through an analogy from a base to a target.

Though there was much work on mechanizing analogy, most of this only partly addressed the issues listed above. Arima [Arima 92] proposed an attempt to answer all the issues at once. Before explaining his idea, we need some preparations for defining terminology.

Analogical reasoning is expressed as the following inference rule:

$$\frac{S(B) \wedge P(B) \quad S(T)}{P(T)}$$

where T represents the *target* object, B the *base* object, S the *similarity property* between T and B, and P the *projected property.*

This inference rule expresses that if we assume an object T is similar to another object B in the sense that they share a common property S then, if B has another property P, we can analogically reason that T also has the same property P. Note that the syntactic similarity of this rule to *modus ponens.* If we generalize the object B to a universally quantified variable X and replace the *and* connective to the *implication* connective, then the first expression of the rule becomes $S(X) \supset P(X)$, thereby the entire rule becomes *modus ponens.*

Arima [Arima 92] tried to link the analogical reasoning to deductive reasoning by modifying the expression $S(B) \wedge P(B)$ to

$$\forall x.(J(x) \wedge S(x) \supset P(x)), \tag{4}$$

where $J(x)$ is a hypothesis added to $S(x)$ in order to logically conclude $P(x)$. If there exists such a $J(x)$, then the analogical reasoning becomes pure deductive reasoning. For example, let us assume that there is a student ($Student_B$) who belongs to an orchestra club and also neglects study. If one happens to know that another student ($Student_T$) belongs to the orchestra club, then we tend to conclude that he also neglects study. The reason why we derive such a conclusion is that we guess that the orchestra club is very active and student members of this busy club tend to neglect study. This reason is an example of the hypothesis mentioned above.

Arima analyzed the syntactic structure of the above $J(x)$ by carefully observing the analogical situation. First, we need to find a proper parameter for the predicate J. Since it is dependent on not only an object but also the similarity property and the projected property, we assume that J has the form of $J(x, s, p)$, where s and p represent the similarity property and the projected property.

From the nature of analogy, we do not expect that there is any direct relationship between the object x and the projected property p. Therefore, the entire $J(x, s, p)$ can be divided into two parts:

$$J(x, s, p) \;=\; J_{att}(s, p) \wedge J_{obj}(x, s), \tag{5}$$

The first component, $J_{att}(s, p)$, corresponds to information extracted from a base. The reason why it does not depend on x comes from the observation that information in the base of the analogy is independent from the choice of an object x.

The second component, $J_{obj}(x, s)$, corresponds to information extracted from the similarity and therefore it does not contain p as its parameter.

Example: Negligent Student

First, let us formally describe the hypothesis described earlier to explain why an orchestra member is negligent of study. It is expressed as follows:

$$\begin{aligned}\forall x, s, p.(\; &Enthusiastic(x, s) \wedge BusyClub(s) \\ &\wedge Obstructive\_to(p, s) \wedge Member\_of(x, s) \\ &\supset Negligent\_of(x, p)\;)\end{aligned} \tag{6}$$

where x, s, and p are variables representing a person, a club and some human activity, respectively. The meaning of each predicate is easy to understand and the explanations are omitted. Since we know that both $Student_B$ and $Student_T$ are members of an orchestra, $Members\_of(X, s)$ corresponds to the similarity property $S(x)$ in (4). On the other hand, since we want to reason the negligence of a student, the projected property $P(x)$ is $Negligent\_of(x, p)$. Therefore, the rest of the expression in (6): $Enthusiastic(x, s) \wedge BusyClub(s) \wedge Obstructive\_to(p, s)$ corresponds to $J(x, s, p)$. From the syntactic feature of this expression, we can conclude that

$$J_{obj}(x, s) = Enthusiastic(x, s),$$
$$J_{att}(s, p) = BusyClub(s) \wedge Obstructive\_to(p, s).$$

The reason why we need J_{obj} is that we are not always aware of an important similarity like *Enthusiastic.* Therefore, we need to infer an important hidden similarity from the given similarity such as $Member\_of$. This inference requires an extra effort in order to apply the above framework of analogy.

The restriction on the syntactic structure of $J(x, s, p)$ is very important since it can be used to prune a search space to access the right base case given the target. This function is particularly important when we apply our analogical inference framework to case based reasoning systems.

5.3 Knowledge Representation

Knowledge representation is one of the central issues in artificial intelligence research. Difficulty arises from the fact that there has been no single knowledge representation scheme for representing various kinds of knowledge while still keeping the simplicity as well as the efficiency of their utilization. Logic was one of the most promising candidates but it was weak in representing structured knowledge and the changing world. Our aim in developing a knowledge representation framework based on logic and logic programming is to solve both of these problems. From the structural viewpoint, we developed an extended relational database which can handle non-normal forms and its corresponding programming language, CRL [Yokota 88a]. This representation allows users to describe their databases in a structured way in the logical framework [Yokota *et al.* 88b].

Recently, we proposed a new logic-based knowledge representation language, Quixote [YasukawaYokota 90]. Quixote follows the ideas developed in CRL and CIL: it inherits object-orientedness from the extended version of CRL and partially specified terms from CIL. One of the main characteristics of the object-oriented features is the notion of object identity. In Quixote, not only simple data atoms but also complex structures are candidates for object identifiers [Morita 90]. Even circular structures can be represented in Quixote. The non-well founded set theory by Aczel [Aczel 88] was adopted to characterize them as a mathematical foundation for such objects, and unification on infinite trees [Colmerauer 82] was adopted as an implementation method.

6 Conclusion

In this article, we summarized the basic research activities of the FGCS project. We emphasized two different directions of logic programming research. One followed logic programming languages where constraint logic programming and concurrent logic programming were focussed. The other followed basic research in artificial intelligence and software engineering based on logic and logic programming.

This project has been like solving a jigsaw puzzle. It is like trying to discover the hidden picture in the puzzle using logic and logic programming as clues. The research problems to be solved were derived naturally from this image. There were several difficult problems. For some problems, we did not even have the right evaluation standard for judging the results. The design of GHC is such an example. Our entire picture of the project helped in guiding our research in the right direction.

The picture is not completed yet. We need further efforts to fill in the remaining spaces. One of the most important parts to be added to this picture is the integration of constraint logic programming and concurrent logic programming. We mentioned our preliminary language/system, GDCC, but this is not yet matured. We need a really useful language which can be efficientlly executed on parallel hardware. Another research subject to be pursued is the realization of a database in KL1. We are actively constructing a parallel database but it is still in the preliminary stages. We believe that there is much affinity between databases and parallelism and we expect a great deal of parallelism from database applications. The third research subject to be pursued is the parallel implementation of abduction and induction. Recently, there has been much work on abduction and induction based on logic and logic programming frameworks. They are expected to provide a foundation for many research themes related to knowledge acquisition and machine learning. Also, both abduction and induction require extensive symbolic computation and, therefore, fit very well with PIM architecture.

Although further research is needed to make our results really useful in a wide range of large-scale applications, we feel that our approach is in the right direction.

Acknowledgements

This paper reflects all the basic research activities in the Fifth Generation Computer Systems project. The author would like to express his thanks to all the researchers in ICOT, as well as those in associated companies who have been working on this project. He especially would like to thank Akira Aiba, Jun Arima, Hiroshi Fujita, Kôiti Hasida, Katsumi Inoue, Noboru Iwayama, Tadashi Kawamura, Ken Satoh, Hiroshi Tsuda, Kazunori Ueda, Hideki Yasukawa and Kazumasa Yokota for their help in greatly improving this work. Finally, he would like to express his deepest thanks to Dr. Fuchi, the director of ICOT, for providing the opportunity to write this paper.

References

[Arima 92] J. Arima, Logical Structure of Analogy. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[Aczel 88] P. Aczel, *Non-Well Founded Set Theory.* CLSI Lecture Notes No. 14, 1988.

[Aiba *et al.* 88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa, Constraint Logic Programming Language CAL. In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.

[BowenKowalski 83] K. Bowen and R. Kowalski, Amalgamating Language and Metalanguage

in Logic Programming. In *Logic Programming,* K. Clark and S. Tarnlund (eds.), Academic Press, 1983.

[ClarkGregory 81] K. L. Clark and S. Gregory, *A Relational Language for Parallel Programming.* In Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, ACM, 1981.

[ClarkGregory 86] K. L. Clark and S. Gregory, *PARLOG: Parallel Programming in Logic.* Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London. Also in *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1, 1986.

[Chikayama 88] T. Chikayama, Programming in ESP – Experiences with SIMPOS –, In *Programming of Future Generation Computers*, Fuchi and Nivat (eds.), North-Holland, 1988.

[Colmerauer 82] A. Colmerauer, Prolog and Infinite Trees. In *Logic Programming,* K. L. Clark and S. Å. Tärnlund (eds.), Academic Press, 1982.

[Colmerauer 86] A. Colmerauer, Theoretical Model of Prolog II. In *Logic Programming and Its Applications*, M. Van Caneghem and D. H. D. Warren (eds.), Albex Publishing Corp, 1986.

[FuchiFurukawa 87] K. Fuchi and K. Furukawa, *The Role of Logic Programming in the Fifth Generation Computer Project.* New Generation Computing, Vol. 5, No. 1, Ohmsha-springer, 1987.

[EshghiKowalski 89] K. Eshghi and R.A. Kowalski, Abduction compared with negation by failure, in: *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, Portugal, 1989.

[Fuchi 90] K. Fuchi, *An Impression of KL1 Programming - from my experience with writing parallel provers -.* In Proc. of KL1 Programming Workshop '90, ICOT, 1990 (in Japanese).

[FujitaFurukawa 88] H. Fujita and K. Furukawa, *A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation.* New Generation Computing, Vol. 6, Nos.2,3, Ohmsha/Springer-Verlag, 1988.

[FujitaHasegawa 91] H. Fujita and R. Hasegawa, *A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm.* In Proc. of the Eighth International Conference on Logic Programming, Paris, 1991.

[Furukawa 92] K. Furukawa, Logic Programming as the Integrator of the Fifth Generation Computer Systems Project, *Communication of the ACM*, Vol. 35, No. 3, 1992.

[Futamura 71] Y. Futamura, Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler. *Systems, Computers, Controls 2*, 1971.

[GelfondLifschitz 88] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, WA, 1988.

[GomardJones 89] C. K. Gomard and N. D. Jones, Compiler Generation by Partial Evaluation: A Case Study. In *Proc. of Information Processing 89*, G. X. Ritter (ed.), North-Holland, 1989.

[Hasida 92] K. Hasida, Dynamics of Symbol Systems - An Integrated Architecture of Cognition. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[HawleyAiba 91] D. Hawley and A. Aiba, *Guarded Definite Clauses with Constraints – Preliminary Report.* Technical Report TR-713, ICOT, 1991.

[Inoue *et al.* 92] K. Inoue, M. Koshimura and R. Hasegawa, Embedding Negation as Failure into a Model Generation Theorem Prover. *To appear in CADE-11: The Eleventh International Conference on Automated Deduction,* Saratoga Springs, NY, June 1992.

[IwayamaSatoh 91] N. Iwayama and K. Satoh, *A Bottom-up Procedure with Top-down Expectation for General Logic Programs with Integrity Constraints.* ICOT Technical Report TR-625, 1991.

[JaffarLassez 86] J. Jaffar and J-L. Lassez, *Constraint Logic Programming.* Technical Report, Department of Computer Science, Monash University, 1986.

[Jones *et al.* 85] N.D. Jones, P. Sestoft, and H. Søndergaard, An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In J-.P. Jouannaud (ed.), *Rewriting Techniques and Applications,* LNCS-202, Springer-Verlag, pp.124–140, 1985.

[Jones *et al.* 88] N. D. Jones, P. Setstoft and H. Sondergaard, MIX: a self-applicable partial evaluator for experiments in compiler generator,Journal of LISP and Symbolic Computation, 1988.

[Kawamura 91] T. Kawamura, *Derivation of Efficient Logic Programs by Synthesizing New Predicates.* Proc. of 1991 International Logic Programming Symposium, pp.611 – 625, San Diego, 1991.

[Koshimura *et al.* 91] M. Koshimura, H. Fujita and R. Hasegawa,
Utilities for Meta-Programming in KL1.
In Proc. of KL1 Programming Workshop'91, ICOT, 1991 (in Japanese).

[Maher 87] M. J. Maher, *Logic semantics for a class of committed-choice programs.* In Proc. of the 4th Int. Conf. on Logic Programming, MIT Press, 1987.

[MantheyBry 88] R. Manthey and F. Bry, *SATCHMO: A Theorem Prover Implemented in Prolog.* In Proc. of CADE-88, Argonne, Illinois, 1988.

[Matsumoto *et al.* 83] Yuji Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa, BUP: A Bottom-up Parser Embedded in Prolog, *New Generation Computing,* Vol. 1, 1983.

[Morita *et al.* 90] Y. Morita, H. Haniuda and K. Yokota, *Object Identity in Quixote.* Technical Report TR-601, ICOT, 1990.

[MukaiYasukawa 85] K. Mukai, and H. Yasukawa, Complex Indeterminates in Prolog and its Application to Discourse Models. *New Generation Computing,* Vol. 3, No. 4, 1985.

[OhsugaSakai 91] A. Ohsuga and K. Sakai, Metis: A Term Rewriting System Generator. In *Software Science and Engineering,* I. Nakata and M. Hagiya (eds.), World Scientific, 1991.

[OkumuraMatsumoto 87] Akira Okumura and Yuji Matsumoto, Parallel Programming with Layered Streams, In *Proc. 1987 International Symposium on Logic Programming,* pp. 224-232, San Francisco, September 1987.

[Plotkin 70] G. D. Plotkin, *A note on inductive generalization.* In B. Meltzer and D. Michie (eds.), Machine Intelligence 5, 1970.

[Poole *et al.* 87] D. Poole, R. Goebel and R. Aleliunas, Theorist: A logical Reasoning System for Defaults and Diagnosis, N. Cercone and G. McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge,* Springer-Verlag, pp.331–352 (1987).

[SakaiAiba 89] K. Sakai and A. Aiba, *CAL: A Theoretical Background of Constraint Logic Programming and its Applications.* J. Symbolic Computation, Vol.8, No.6, pp.589-603, 1989.

[Saraswat 89] V. Saraswat, *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1989.

[SatohIwayama 92] K. Satoh and N. Iwayama, A Correct Top-down Proof Procedure for a General Logic Program with Integrity Constraints. In *Proc. of the 3rd International Workshop on Extensions of Logic Programming,* E. Lamma and P. Mello (eds.), Facalta di Ingegneria, Universita di Bologna, Italy, 1992.

[SekiFurukawa 87] H. Seki and K. Furukawa, *Notes on Transformation techniques for Generate and Test Logic Programs.* In Proc. 1987 Symposium on Logic Programming, IEEE Computer Society Press, 1987.

[Shapiro 83] E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter.* Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.

[Sugimura *et al.* 88] R. Sugimura, K. Hasida, K. Akasaka, K. Hatano, Y. Kubo, T. Okunishi, and T. Takizuka, A Software Environment for Research into Discourse Understanding Systems. In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.

[TakeuchiFurukawa 86] A. Takeuchi and K. Furukawa, *Partial Evaluation of Prolog Programs*

and Its Application to Meta Programming. In Proc. IFIP'86, North-Holland, 1986.

[Taki 88] K. Taki, *The Parallel Software Research and Development Tool: Multi-PSI system.* In Programming of Future Generation Computers, K. Fuchi and M. Nivat (eds.), North-Holland, 1988.

[Taki 89] K. Taki, *The FGCS Computing Architecture.* In Proc. IFIP'89, North-Holland, 1989.

[TanakaYoshioka 88] Y. Tanaka, and T. Yoshioka, Overview of the Dictionary and Lexical Knowledge Base Research. In *Proc. FGCS'88*, Tokyo, 1988.

[Tsuda 92] H. Tsuda, cu-Prolog for Constraint-based Grammar. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[Ueda 86a] K. Ueda, *Guarded Horn Clauses.* In Logic Programming '85, E. Wada (ed.), Lecture Notes in Computer Science, 221, Springer-Verlag, 1986.

[Ueda 86b] K. Ueda, *Making Exhaustive Search Programs Deterministic.* In Proc. of the Third Int. Conf. on Logic Programming, Springer-Verlag, 1986.

[UedaChikayama 90] K. Ueda and T. Chikayama, *Design of the Kernel Language for the Parallel Inference Machine.* The Computer Journal, Vol. 33, No. 6, pp. 494–500, 1990.

[Warren 83] D. H. D. Warren, *An Abstract Prolog Instruction Set.* Technical Note 304, Artificial Intelligence Center, SRI, 1983.

[YasukawaYokota 90] H. Yasukawa and K. Yokota, *Labeled Graphs as Semantics of Objects.* Technical Report TR-600, ICOT, 1990.

[Yokota 88a] K. Yokota, *Deductive Approach for Nested Relations.* In Programming of Future Generation Computers II, K. Fuchi and L. Kott (eds.), North-Holland, 1988.

[Yokota *et al.* 88b] K. Yokota, M. Kawamura and A. Kanaegami, Overview of the Knowledge Base Management System(KAPPA). In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Summary of the Parallel Inference Machine and its Basic Software

Shunichi Uchida

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
uchida@icot.or.jp

Abstract

This paper aims at a concise introduction to the PIM and its basic software, including the overall framework of the project. Now an FGCS prototype system is under development. Its core is called a parallel inference system which includes a parallel inference machine, PIM, and its operating system, PIMOS. The PIM includes five hardware modules containing about 1,000 element processors in total. On the parallel inference system, there is a knowledge base management system (KBMS). The PIMOS and KBMS make a software layer called a basic software of the prototype system. These systems are already being run on the PIM. On these systems, a higher-level software layer is being developed. It is called a knowledge programming software. This is to be used as a tool for more powerful inference and knowledge processing. It contains language processors for constraint logic programming languages, parallel theorem provers and natural language processing systems. Several experimental application programs are also being developed for both general evaluation of the PIM and the exploration of new application fields for knowledge processing. These achievements with the PIM and its basic software easily surpass the research targets set up at the beginning of the project.

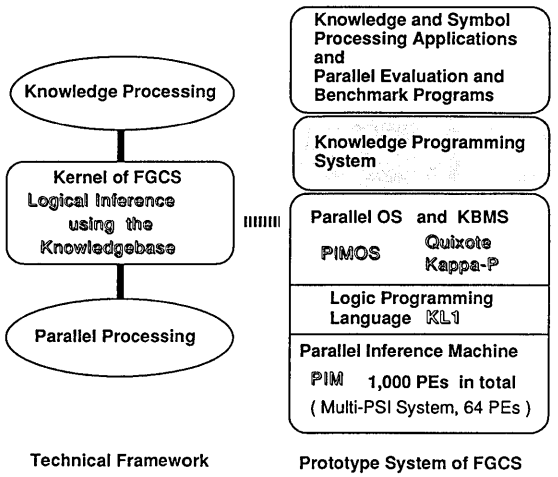


Figure 1: Framework of FGCS Project

1 Introduction

Since the fifth generation computer systems project (FGCS) was started in June, 1982, 10 years have passed, and the project is approaching its goal. This project assumed that "logic" was the theoretical backbone of future knowledge information processing, and adapted logic programming as the kernel programming language of fifth generation computer systems. In addition to the adaptation of logic programming, highly parallel processing for symbolic computation was considered indispensable for implementing practical knowledge information processing systems. Thus, the project aimed to create a new computer technology combining knowledge processing with parallel processing using logic programming.

Now an FGCS prototype system is under development. This system integrates the major research achievements of these 10 years so that they can be evaluated and demonstrated. Its core is called **a parallel inference system** which includes a parallel inference machine, PIM, and its operating system, PIMOS. The PIM includes five hardware modules containing about 1,000 element processors in total. It also includes a language processor for a parallel logic language, KL1.

On the parallel inference system, there is **a knowledge base management system (KBMS)**. The KBMS includes a database management system (DBMS), Kappa-P, as its lower layer. The KBMS provides a knowledge representation language, Quixote,

based on the deductive (and) object-oriented database. The PIMOS and KBMS make a software layer called **a basic software** of the prototype system. These systems are already being run on the PIM. The PIM and basic software are now being used as a new research platform for building experimental parallel application programs. They are the most complete of their kind in the world.

On this platform, a higher-level software layer is being developed. This is to be used as a tool for more powerful inference and knowledge processing. It contains language processors for constraint logic programming languages, parallel theorem provers, natural language processing systems, and so on. These software systems all include the most advanced knowledge processing techniques, and are at the leading edge of advanced software science.

Several experimental application programs are also being developed for both general evaluation of the PIM and the exploration of new application fields for knowledge processing. These programs include a legal reasoning system, genetic information processing systems, and VLSI CAD systems. They are now operating on the parallel inference system, and indicate that parallel processing of knowledge processing applications is very effective in shortening processing time and in widening the scope of applications. However, they also indicate that more research should be made into parallel algorithms and load balancing methods for symbol and knowledge processing. These achievements with the PIM and its basic software easily surpass the research targets set up at the beginning of the project.

This paper aims at a concise introduction to the PIM and its basic software, including the overall framework of the project. This project is the first Japanese national project that aimed at making a contribution to world computer science and the promotion of international collaboration. We have published our research achievements wherever possible, and distributed various programs from time to time. Through these activities, we have also been given much advice and help which was very valuable in helping us to attain our research targets. Thus, our achievements in the project are also the results of our collaboration with world researchers on logic programming, parallel processing and many related fields.

2 Research Targets and Plan

2.1 Scope of R & D

The general target of the project is the development of a new computer technology for knowledge information processing.

Having "mathematical logic" as its theoretical backbone, various research and development themes were established on software and hardware technologies focusing on knowledge and symbol processing. These themes are grouped into the following three categories:

2.1.1 Parallel inference system

The core portion of the project was the research and development of the parallel inference system which contains the PIM, a KL1 language processor, and the PIMOS. To make the goal of the project clear, a FGCS prototype system was considered a major target. This was to be build by integrating many experimental hardware and software components developed around logic programming.

The prototype system was defined as a parallel inference system which is intended to have about 1,000 element processors and attain more than 100M LIPS (Logical Inference Per Second) as its execution speed. It was also intended to have a parallel operating system, PIMOS, as part of the basic software which provides us with an efficient parallel programming environment in which we can easily develop various parallel application programs for symbol and knowledge processing, and run them efficiently. Thus, this is regarded as the development of a super computer for symbol and knowledge processing.

It was intended that overall research and development activities would be concentrated so that the major research results could be integrated into a final prototype system, step by step, over the timespan allotted to the project.

2.1.2 KBMS and knowledge programming software

Themes in this category aimed to develop a basic software technology and theory for knowledge processing.

- Knowledge representation and knowledge base management
- High-level problem solving and inference software
- Natural language processing software

These research themes were intended to create new theories and software technologies based on mathematical logic to describe various **knowledge fragments** which are parts of **"natural" knowledge bases** produced in our social systems. We also intended to store them in a computer system as components of **"artificial" knowledge bases** so that they can be used to build various intelligent systems.

To describe the knowledge fragments, a knowledge representation language has to be provided. It can be regarded as a very high-level programming language executed by a sophisticated inference mechanism which is much cleverer than the parallel inference system. Natural language processing research is intended to cover

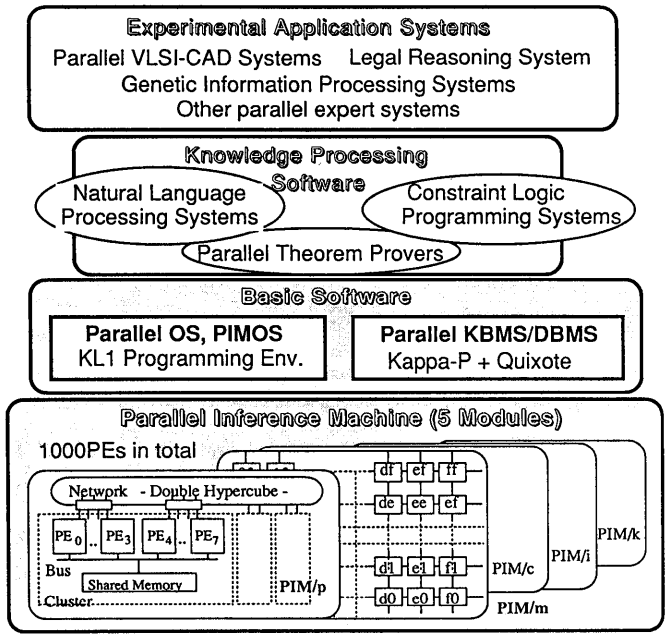


Figure 2: Organization of Prototype System

research on knowledge representation methods and such inference mechanisms, in addition to research on easy-to-use man-machine interface functions. Experimental software building for some of these research themes was done on the sequential inference machines because the level of research was so basic that computational power was not the major problem.

2.1.3 Benchmarking and evaluation systems

- Benchmarking software for the parallel inference system
- Experimental parallel application software

To carry out research on an element technology in computer science, it is essential that an experimental software system is built. Typical example problems can then be used to evaluate theories or methods invented in the progress of the research.

To establish general methods and technologies for knowledge processing, experimental systems should be developed for typical problems which need to process knowledge fragments as sets of rules and facts.

These problems can be taken from engineering systems, including machine design and the diagnosis of machine malfunction, or from social systems such as medical care, government services, and company management.

Generally, the exploitation of computer technology for knowledge processing is far behind that for scientific calculation. Recent expert systems and machine translation systems are examples of the most advanced knowledge processing systems. However, the numbers of rules and facts in their knowledge bases are several hundreds on average.

This scale of knowledge base may not be large enough to evaluate the maximum power of parallel inference system having about 1,000 element processors. Thus, research and development on large-scale application systems is necessary not only for knowledge processing research but also for the evaluation of the parallel inference system. Such application systems should be widely looked for in many new fields.

The scope of research and development in this project is very wide, however, the parallel inference system is central to the whole project. It is a very clear research target. Software research and development should also cover diverse areas in recent software technology. However, it has "logic" as the common backbone.

It was also intended that major research achievements should be integrated into one prototype system. This has made it possible for us to organize all of our research and development in a coherent way. At the beginning of the project, only the parallel inference machine was defined as a target which was described very clearly. The other research targets described above were not planned at the

beginning of the project. They have been added in the middle of the intermediate stage or at the final stage.

2.2 Overall R & D plan

After three years of study and discussions on determining our major research fields and targets, the final research and development plan was determined at the end of fiscal 1981 with the budget for the first fiscal year.

At that time, practical logic programming languages had begun to be used in Europe mainly for natural language processing. The feasibility and potential of logic languages had not been recognized by many computer scientists. Thus, there was some concern that the level of language was too high to describe an operating system, and that the overhead of executing logic programs might be too large to use it for practical applications. This implies that research on logic programming was in its infancy.

Research on parallel architectures linked with high-level languages was also in its infancy. Research on dataflow architectures was the most advanced at that time. Some dataflow architecture was thought to have the potential for knowledge and symbol processing. However, its feasibility for practical applications had not yet been evaluated.

Most of the element technologies necessary to build the core of the parallel inference system were still in their infancy. We then tried to define a detailed research plan step by step for the 10-year project period. We divided the 10-year period into three stages, and defined the research to be done in each stage as follows:

- **Initial stage (3 years) :**
  -Research on potential element technologies
  -Development of research tools

- **Intermediate stage (4 years) :**
  -First selection of major element technologies for final targets
  -Experimental building of medium-scale systems

- **Final stage (3 years) :**
  -Second selection of major element technologies for final targets
  -Experimental building of a final full-scale system

At the beginning of the project, we made a detailed research and development plan only for the initial stage. We decided to make detailed plans for the intermediate and final stages at the end of the stage before, so that the plans would reflect the achievements of the previous stage. The research budget and manpower were to be decided depending on the achievements. It was likely that the project would effectively be terminated at the end of the initial stage or the intermediate stage.

3 Inference System in the Initial Stage

3.1 Personal Sequential Inference Machine (PSI-I)

To actually build the parallel inference system, especially a productive parallel programming environment which is now provided by PIMOS, we needed to develop various element technologies step by step to obtain hardware and software components. On the way toward this development, the most promising methods and technologies had to be selected from among many alternatives, followed by appropriate evaluation processes. To make this selection reliable and successful, we tried to build experimental systems which were as practical as possible.

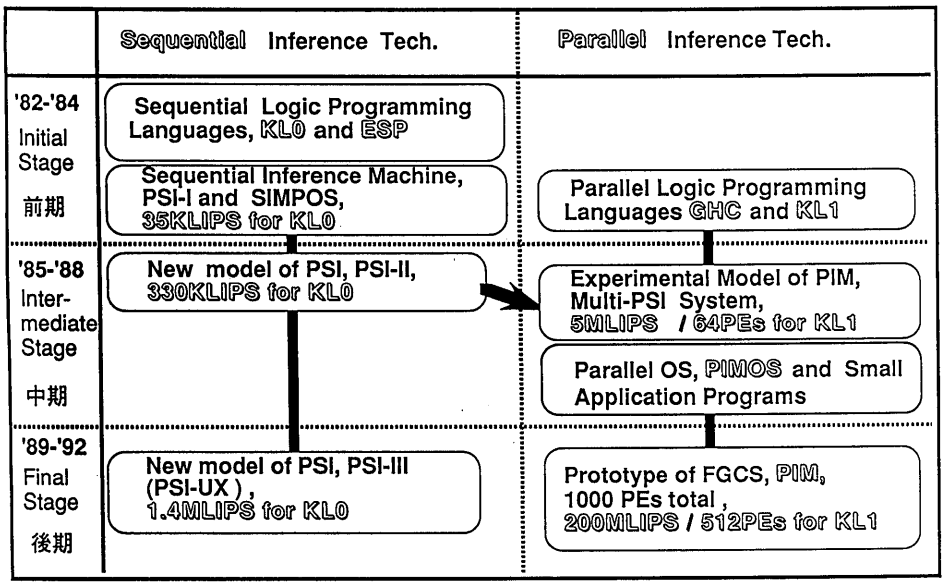
In the initial stage, to evaluate the descriptive power and execution speed of logic languages, a personal sequential machine, PSI, was developed. This was a logic programming workstation. This development was also aimed at obtaining a common research tool for software development. The PSI was intended to attain an execution speed similar to DEC10 Prolog running on a DEC20 system, which was the fastest logic programming system in the world.

To begin with, a PSI machine language, **KL0**, was designed based on Prolog. Then a hardware system was designed for the KL0. We employed tag architecture for the hardware system. Then we designed a system description language, **ESP**, which is a logic language having a class and inheritance mechanisms to make program modules efficiently.[Chikayama 1984] ESP was used not only to write the operating system for PSI, which is named **SIMPOS**, but also to write many experimental software systems for knowledge processing research.

The development of the PSI machine and SIMPOS was successful. We were impressed by the very high software productivity of the logic language. The execution speed of the PSI was about 35K LIPS and exceeded its target. However, we realized that we could improve its architecture by using the optimization capability of a compiler more effectively. We produced about 100 PSI machines to distribute as a common research tool. This version of the PSI is called PSI-I.

In conjunction with the development of PSI-I and SIMPOS, research on parallel logic languages was actively pursued. In those days, pioneering efforts were being made on parallel logic languages such as PARLOG and Concurrent Prolog. [Clark and Gregory 1984], [Shapiro 1983] We learned much from this pioneering research, and aimed to obtain a simpler language more suited for a machine language for a parallel inference machine. Near the end of the initial stage, a new parallel logic language, **GHC** was designed. [Ueda 1986]

Table 1: Development of Inference Systems



3.2 Effect of PSI development on the research plan

The experience gained in the development of PSI-I and SIMPOS heavily affected the planning of the intermediate stage.

3.2.1 Efficiency in program production

One of the important questions related to logic language was the feasibility of writing an operating system which needs to describe fine detailed control mechanisms. Another was its applicability to writing large-scale programs. SIMPOS development gave us answers to these questions. The SIMPOS has a multi-window-based user interface, and consists of more than 100,000 ESP program lines. It was completed by a team of about 20 software researchers and engineers over about two years. Most of the software engineers were not familiar with logic languages at that time.

We found that logic languages have much higher productivity and maintainability than conventional von Neumann languages. This was obvious enough to convince us to describe a parallel operating system also in a logic language.

3.2.2 Execution performance

The PSI-I hardware and firmware attained about 35K LIPS. This execution speed was sufficient for most knowledge processing applications. The PSI had an 80 MB main memory. It was a very big memory compared to mainframe computers at that time. We found that this large memory and fast execution speed made a logic language a practical and highly productive tool for software prototyping.

The implementation of the PSI-I hardware required 11 printed circuit boards. As the amount of hardware became clear, we established that we could obtain an element processor for a parallel machine if we used VLSI chips for implementation.

For the KL0 language processor which was implemented in the firmware, we estimated that better optimization of object code made by the compiler would greatly improve execution speed. (Later, this optimization was made by introducing of the "WAM" code.[Warren 1983])

The PSI-I and SIMPOS proved that logic languages are a very practical and productive vehicle for complex knowledge processing applications.

4 Inference Systems in the Intermediate Stage

4.1 A parallel inference system

4.1.1 Conceptual design of KL1 and PIMOS

The most important target in the intermediate stage was a parallel implementation of a KL1 language processor, and the development of a parallel operating system, PIMOS.

The full version of GHC, was still too complex for the machine implementation. A simpler version, **FGHC**, was designed.[Chikayama and Kimura 1985] Finally, a practical parallel logic language, KL1, was designed based on FGHC.

The KL1 is a parallel language classified as an

AND-parallel logic programming language. Its language processor includes an automatic memory management mechanism and a dataflow process synchronization mechanism. These mechanisms were considered essential for writing and compiling large parallel programs. The first problem was whether they could be implemented efficiently. The second problem was what kind of firmware and hardware support would be possible and effective.

In addition to problems in implementing the KL1 language processor, the design of PIMOS created several important problems. The role of PIMOS is different from that of conventional operating systems. PIMOS does not need to do primary process scheduling and memory management because these tasks are performed by the language processor. It still has to perform resource management for main memory and element processors, and control the execution of user programs. However, a much more difficult role was added. It must allow a user to divide a job into parallel processable processes and distribute them to many element processors. Processor loads must be well balanced to attain better execution performance. In knowledge and symbol processing applications, the dynamic structure of a program is not regular. It is difficult to estimate the dynamic program structure. It was desirable that PIMOS could offer some support for efficient job division and load balancing problems.

These problems in the language processor and the operating system were very new, and had not been studied as practical software problems. To solve these problems, we realized that we must have appropriate parallel hardware as a platform to carry out practical software experiments using a trial and error.

4.1.2 PSI-II and Multi-PSI system

In conjunction with the development of KL1 and PIMOS, we needed to extend our research and develop new theories and software technologies for knowledge processing using logic programming. This research and development demanded improvement of PSI-I machines in such aspects as performance, memory size, cabinet size, disk capacity, and network connection.

We decided to develop a smaller and higher-performance model of PSI, to be called **PSI-II**. This was intended to provide a better workstation for use as a common tool and also to obtain an element processor for the parallel hardware to be used as a platform for parallel software development. This hardware was called **a multi-PSI system**. It was regarded as a small-scale experimental version of the PIM. As many PSI-II machines were produced, we anticipated having very stable element processors for the multi-PSI system.

The PSI-II used VLSI gate array chips for its CPU. The size of the cabinet was about one sixth that of PSI-I. Its execution speed was 330K LIPS, about 10 times faster than that of PSI-I. This improvement was attained mainly through employment of the better compiler optimization technique and improvement of its machine architecture. The main memory size was also expanded to 320 MB so that prototyping of large applications could be done quickly.

In the intermediate stage, many experimental systems were built on PSI-I and PSI-II systems for knowledge processing research. These included small-to-medium scale expert systems, a natural language discourse understanding system, constraint logic programming systems, a database management system, and so on. These systems were all implemented in the ESP language using about 300 PSI-II machines distributed to the researchers as their personal tools.

The development of the multi-PSI system was completed in the spring of 1988. It consists of 64 element processors which are connected by an 8 by 8 mesh network. One element processor is contained in three printed circuit boards. Eight element processors are contained in one cabinet. Each element processor has an 80 MB main memory. Thus, a multi-PSI was to have about 5GB memories in total. This hardware was very stable, as we had expected. We produced 6 multi-PSI systems and distributed them to main research sites.

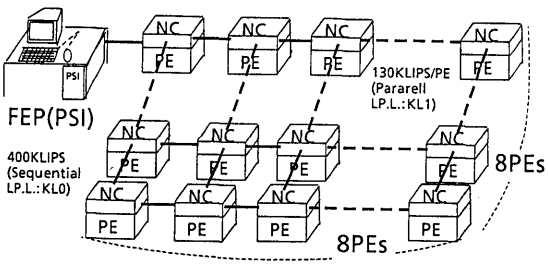
4.1.3 KL1 language processor and PIMOS

This was the first trial implementation of a distributed language processor of a parallel logic language, and a parallel operating system on real parallel hardware, used as a practical tool for parallel knowledge processing applications.

The KL1 distributed language processor was an integration of various complex functional modules such as a distributed garbage collector for loosely-coupled memories. The automatic process synchronization mechanism based on the dataflow model was also difficult to implement over the distributed element processors. Parts of these mechanisms had to be implemented combined with some PIMOS functions such as a dynamic on-demand loader for object program codes. Other important functions related to the implementation of the language processor were support functions like system debugging, system diagnostic, and system maintenance functions.

In addition to these functions for the KL1 language processor, many PIMOS functions for resource management and execution control had to be designed and implemented step by step, with repeated partial module building and evaluation.

This partial module building and evaluation was done for core parts of the KL1 language processor and PIMOS, using not only KL1 but also ESP and C languages. An appropriate balance between the functions of the language processor and the functions of PIMOS was considered. The language processor was implemented in a PSI-II firmware for the first time. It worked as a pseudo parallel simulator of KL1, and was used as a PIMOS



- Machine language: KL1-b
- Max. 64PEs and two FEPs (PSI-II) connected to LAN
- Architecture of PE:
  - Microprogram control (64 bits/word)
  - Machine cycle: 200ns, Reg.file: 64W
  - Cache: 4 KW, set associative/write-back
  - Data width: 40 bits/word
  - Memory capacity: 16MW (80MB)
- Network:
  - 2-dimensional mesh
  - 5MB/s x 2 directions/ch with 2 FIFO buffers/ch
  - Packet routing control function

Figure 3: Multi-PSI System: Main features and Appearance

development tool. It was eventually extended and transported to the multi-PSI system.

In the development of PIMOS, the first partial module building was done using the C language in a Unix environment. This system is a tiny subset of the KL1 language processor and PIMOS, and is called the PIMOS Development Support System (PDSS). It is now distributed and used for educational purposes. The first version of PIMOS was released on the PSI-II with the KL1 firmware language processor. This is called **a pseudo multi-PSI system**. It is currently used as a personal programming environment for KL1 programs.

With the KL1 language processor fully implemented in firmware, one element processor or a PSI-II attained about 150 KLIPS for a KL1 program. It is interesting to compare this speed with that for a sequential ESP program. As a PSI-II attains about 300 KLIPS for a sequential ESP program, the overhead for KL1 caused by automatic process synchronization halves the execution speed. This overhead is compensated for by efficient parallel processing. A full-scale multi-PSI system of 64 element processors could attain 5 - 10 MLIPS. This speed was considered sufficient for the building of experimental software for symbol and knowledge processing applications. On this system, simple benchmarking programs and applications such as puzzle programs, a natural language parser and a Go-game program were quickly developed. These programs and the multi-PSI system was demonstrated in FGCS'88.[Uchida et al. 1988] These proved that KL1 and PIMOS could be used as a new platform for parallel software research.

4.2 Overall design of the parallel inference system

4.2.1 Background of the design

The first question related to the design of the parallel inference system was what kind of functions must be provided for modeling and programming complex problems, and for making them run on large-scale parallel hardware.

When we started this project, research on parallel processing still tended to focus on hardware problems. The major research and development interest was in SIMD or MIMD type machines applied for picture processing or large-scale scientific calculations. Those applications were programmed in Fortran or C. Control of parallel execution of those programs, such as job division and load balancing, was performed by built-in programs or prepared subroutine libraries, and could not be done by ordinary users.

Those machines excluded most of the applications which include irregular computations and require general parallel programming languages and environments. This tendency still continues. Among these parallel machines, some dataflow machines were exceptional and had the potential to have functional languages and their general parallel programming environment.

We were confident that a general parallel programming

language and environment is indispensable for writing parallel programs for large-scale symbol and knowledge processing applications, and that they must provide such functions as follows:

1. An automatic memory management mechanism for distributed memories (parallel garbage collector)
2. An automatic process synchronization mechanism based on a dataflow scheme
3. Various support mechanisms for attaining the best job division and load balancing.

The first two are to be embedded in the language processor. The last is to be provided in a parallel operating system. All of these answer the question of how to write parallel programs and map them on parallel machines.

This mapping could be made fully automatic if we limited our applications to very regular calculations and processing. However, for the applications we intend, the mapping process, which includes job division and load-balancing, should be done by programmers using the functions of the language processor and operating system.

4.2.2 A general parallel programming environment

Above mechanisms for mapping should be implemented in the following three layers:

1. A parallel hardware system consisting of element processors and inter-connection network (PIM hardware)
2. A parallel language processor consisting of run-time routines, built-in functions, compilers and so on (KL1 language processor)
3. A parallel operating system including a programming environment (PIMOS)

At the beginning of the intermediate stage, we tried to determine the roles of the hardware, the language processor and the operating system. This was really the start of development.

One idea was to aim at hardware with many functions and using high density VLSI technology, as described in early papers on dataflow machine research. It was a very challenging approach. However, we thought it too risky because changes to the logic circuits in VLSI chips would have a long turn-around time even if the rapid advance of VLSI technology was taken into account. Furthermore, we thought it would be difficult to run hundreds of sophisticated element processors for a few days to a few weeks without any hardware faults.

Implementation of the language processor and the operating system was thought to be very difficult too. As there were no prior examples, we could not make any reliable quantitative estimation of the overhead caused by these software systems. This implementation was therefore considered risky too.

Finally, we decided not to make an element processor too complex , so that our hardware engineers could provide the software researchers with a large-scale hardware platform stable enough to make the largest-scale software experiments in the world.

However, we tried to add cost-effective hardware support for KL1 to the element processor, in order to attain a higher execution speed. We employed tag architecture to support the automatic memory management mechanism as well as faster execution of KL1 programs. The automatic synchronization mechanism was to be implemented in firmware. The supports for job division and load balancing were implemented partially by the firmware as primitives of the KL1 language, but they were chiefly implemented by the operating system. In a programming environment of the operating system, we hoped to provide a semi-automatic load balancing mechanism as an ultimate research goal.

PIMOS and KL1 hide from users most of the architectural details of the element processors and network system of PIM hardware. A parallel program is modeled and programmed depending on a parallel model of an application problem and algorithms designed by a programmer. The programmer has great freedom in dividing programs because a KL1 program is basically constructed from very fine-grain processes.

As a second step, the programmer can decide the grouping of fine-grain processes in order to obtain **an appropriate granularity** as divided jobs, and then specify how to dispatch them to element processors using a special notation called "pragma". This two step approach in parallel programming makes it easy and productive.

We decided to implement the memory management mechanism and the synchronization mechanism mainly in the firmware. The job division and load balancing mechanism was to be implemented in the software. We decided not to implement uncertain mechanisms in the hardware.

The role of the hardware system was to provide a stable platform with enough element processors, execution speed, memory capacity, number of disks and so on. The demands made on the capacity of a cache and a main memory were much larger than those of a general purpose microprocessor of that time. The employment of tag architecture contributed to the simple implementation of the memory management mechanism and also increased the speed of KL1 program execution.

5 R & D in the final stage

5.1 Planning of the final stage

At the end of the intermediate stage, an experimental medium-scale parallel inference system consisting of the multi-PSI system, the KL1 language processor, and PIMOS was successfully completed. On this system, several small application programs were developed and run efficiently in parallel. This proved that symbol and knowledge processing problems had sufficient parallelism and could be written in KL1 efficiently. This success enabled us to enter the final stage.

Based on research achievements and newly developed tools produced in the intermediate stage, we made a detailed plan for the final stage. One general target was to make a big jump from the hardware and software technologies for the multi-PSI system to the ones for the PIM, with hundreds of element processors. Another general target was to make a challenge for parallel processing of large and complex knowledge processing applications which had never been tackled anywhere in the world, using KL1 and the PIM.

Through the research and development directed to these targets, we expected that a better parallel programming methodology would be established for logic programming. Furthermore, the development of large and complex application programs would not only encourage us to create new methods of building more intelligent systems systematically but could also be used as practical benchmarking programs for the parallel inference system. We intended to develop new techniques and methodologies.

1. Efficient parallel software technology

   (a) Parallel modeling and programming techniques
   -Parallel programming paradigms
   -Parallel algorithms

   (b) Efficient mapping techniques of parallel processes to parallel processors
   -Dynamic load balancing techniques
   -Performance debugging support

2. New methodologies to build intelligent systems using the power of the parallel inference system

   (a) Development of a higher-level reasoning or inference engine and higher-level programming languages

   (b) Methodologies for knowledge representation and knowledge base management (methodology for knowledge programming)

The research and development themes in the final stage were set up as follows:

1. PIM hardware development

   We intended to build several models with different architectures so that we could compare mapping problems between the architectures and program models. The number of element processors for all the modules was planned about 1,000.

2. The KL1 language processor for the PIM modules

   We planned to develop new KL1 language processors which took the architectural differences on the PIM modules into account.

3. Improvement and extension of PIMOS

   We intended to develop an object-oriented language, AYA, over KL1, a parallel file system, and extended performance debugging tools for its programming environment.

4. Parallel DBMS and KBMS

   We planned to develop a parallel and distributed database management system, using several disk drives connected to PIM element processors, was intended to attain high throughput and consequently a high information retrieval speed. As we had already developed a data base management system, Kappa-II, which employed a nested relational model on the PSI machine, we decided to implement a parallel version of Kappa-II. However, we redesiged its implementation, employing the distributed database model and using KL1. This parallel version is called Kappa-P. We plan to develop a knowledge base management system on the Kappa-P. This would be based on the deductive object-oriented DB, having a knowledge representation language, Quixote.

5. Research on knowledge programming software

   We intended to continue various basic research activities to develop new theories, methodologies and tools for building knowledge processing application systems. These activities were grouped together as research on knowledge programming software.

   This included research themes such as a parallel constraint logic programming language, mathematical systems including theorem provers, natural language processing systems such as a grammar design system, and an intelligent sentence generation system for man-machine interfacing.

6. Benchmarking and experimental parallel application systems

   To evaluate the parallel inference system and the various tools and methodologies developed in the above themes, we decided to make more effort to

explore new applications of parallel knowledge processing. We began research into a legal expert system, a genetic information processing systems and so on.

5.2 R & D results in the final stage

The actual research activities into the themes described above differed according to characteristics. In the development of the parallel inference system, we focused on the integration of PIM hardware and some software components. In our research on knowledge programming software, we continued basic research and experimental software building to create new theories and develop parallel software technologies for the future.

5.2.1 PIM hardware and KL1 language processor

A role of the PIM hardware was to provide software researchers with an advanced platform which would allow large-scale software development for knowledge processing.

Another role was to obtain various evaluation data in the architecture and hardware structure of the element processors and network systems. In particular, we wanted to analyze the performance of large-scale parallel programs on various architectures (machine instruction sets) and hardware structures, so that hardware engineers could design more powerful and cost-effective parallel hardware in the future.

In the conceptual design of the PIM hardware, we realized that there were many alternative designs for the architecture of an element processor and the structure of a network system. For the architecture of an element processor, we could choose between a CISC type instruction set implemented in firmware and a RISC type instruction set. On the interconnection network, there were several opinions, including a two dimensional mesh network like the multi-PSI, a cross-bar switch, and a common bus and coherent cache.

To design the best hardware, we needed to find out the mapping relationships between program behavior and the hardware architectures and structures. We had to establish criteria for the design of the parallel hardware, reflecting the algorithms and execution structures of application programs.

To gather the basic data we needed to obtain this design criteria, we tried to categorize our design choices into five groups and build five PIM modules. The main features of these five modules are listed in Table 2. The number of element processor required for each module was determined depending on the main purpose of the module. Large modules have 256 to 512 element processors, and were intended to be used for software experiments. Small modules have 16 or 20 element processors and were built for architectural experiments and evaluation.

All of these modules were designed to support KL1 and PIMOS, so that software researchers could run one program on the different modules and compare and analyze the behaviors of parallel program execution.

A PIM/m module employed architecture similar to the multi-PSI system. Thus, its KL1 language processor could be developed by simply modifying and extending that of the multi-PSI system. For other modules, namely PIM/p, PIM/c, PIM/k, and PIM/i, the KL1 language processor had to be newly developed because all of these modules have a cluster structure. In a cluster, four to eight element processors were tightly coupled by a shared memory and a common bus with coherent caches. While communication between element processors is done through the common bus and shared memory, communication between clusters is done via a packet switching network. These four PIM modules have different machine instruction sets.

We intended to avoid the duplication of development work for the KL1 language processor. We used the KL1-C language to write PIMOS and the usual application programs. A KL1-C program is compiled into the KL1-B language, which is similar to the "WAM" as shown in Figure 5. We defined an additional layer between the KL1-B language and the real machine instruction. This layer is called the virtual hardware layer. It has a virtual machine instruction set called "PSL". The specification of the KL1-B interpreter is described in PSL. This specification is semi-automatically converted to a real interpreter or runtime routines dedicated to each PIM modules. The specification in PSL is called a virtual PIM processor (the VPIM processor for short) and is common to four PIM modules.

PIM/p, PIM/m and PIM/c are intended to be used for large software experiments; the other modules were intended for architectural evaluations. We plan to produce a PIM/p with 512 element processors, and a PIM/m with 384 element processors. Now, at the beginning of March 1992, a PIM/m of 256 processors has just started to run a couple of benchmarking programs.

We aimed at a processing speed of more than 100 MLIPS for the PIM modules. The PIM/m with 256 processors will attain more than 100 MLIPS as its peak performance. However, for a practical application program, this speed may be much reduced, depending on the characteristics of the application program and the programming technique. To obtain better performance, we must attempt to augment the effect of compiler optimization and to implement a better load balancing scheme. We plan to run various benchmarking programs and experimental application programs to evaluate the gain and loss of implemented hardware and software functions.

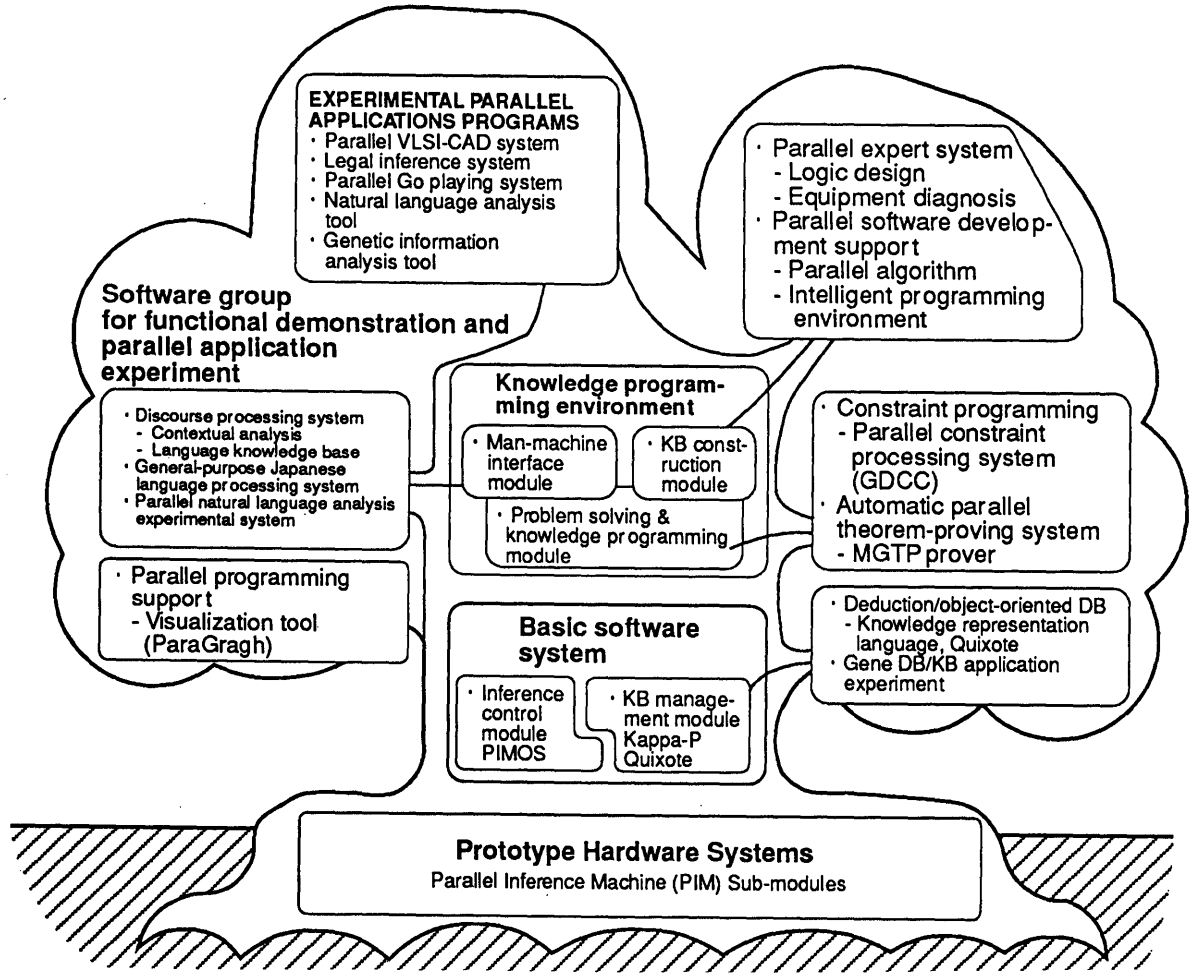


Figure 4: Research Themes in the Final Stage

Table 2: Features of PIM modules

Item	PIM/p	PIM/c	PIM/m	PIM/i	PIM/k
Machine instructions	RISC-type + macro instructions	Horizontal microinstructions	Horizontal microinstructions	RISC-type	RISC-type
Target cycle time	60 nsec	65 nsec	50 nsec	100 nsec	100 nsec
LSI devices	Standard cell	Gate array	Cell base	Standard cell	Custom
Process Technology (line width)	0.96 μm	0.8 μm	0.8 μm	1.2 μm	1.2 μm
Machine configuration	Multicluster connections (8 PEs linked to a shared memory) in a hypercube network	Multicluster connections (8 PEs + CC linked to a shared memory) in a crossbar network	Two-dimensional mesh network connections	Shared memory connections through a parallel cache	Two-level parallel cache connections
Number of PEs connected	512 PEs	256 PEs	256 PEs	16 PEs	16 PEs

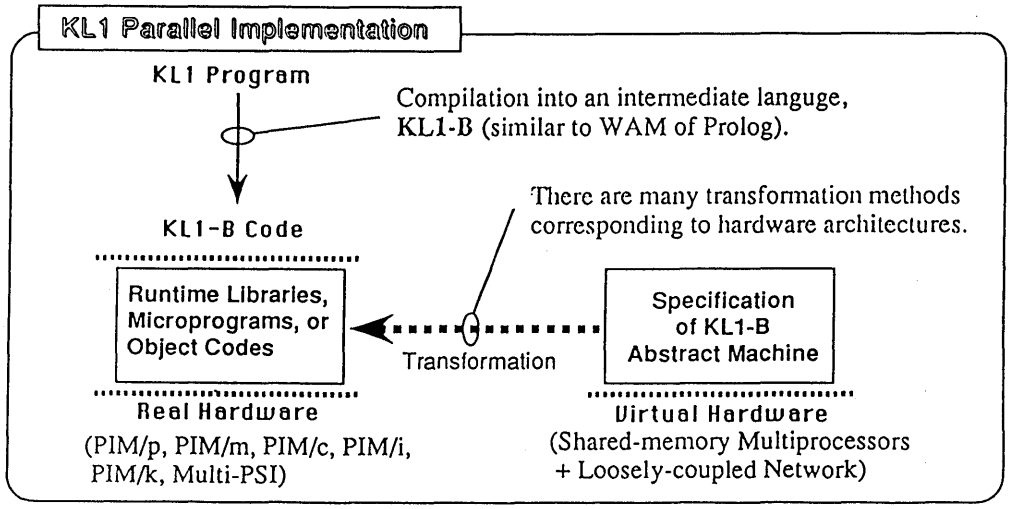
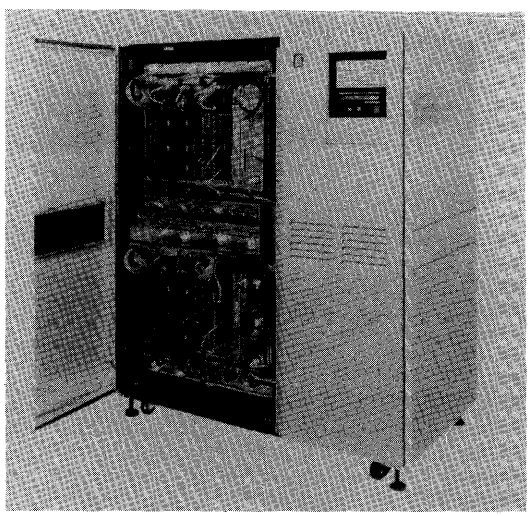
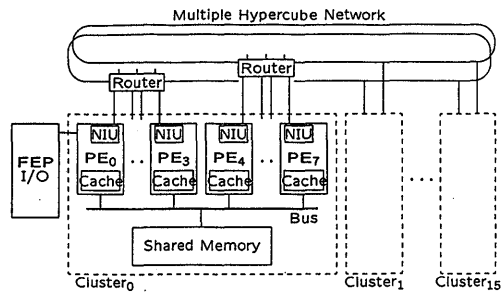


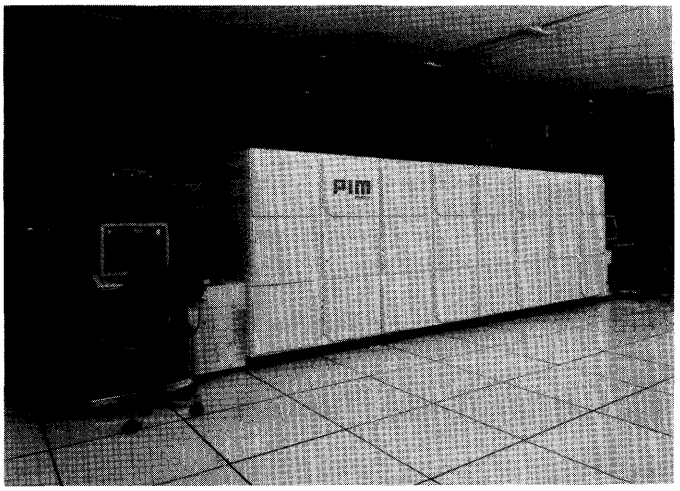
Figure 5: KL1 Language Processor and VPIM

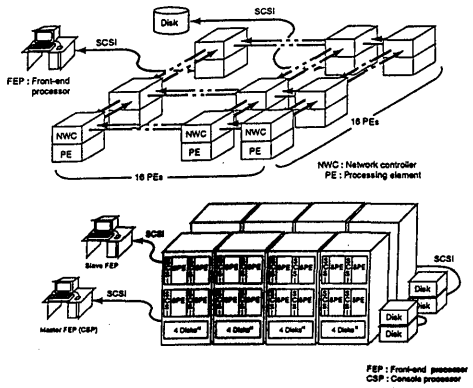




- Machine language: KL1-b
- Architecture of PE and cluster
  - RISC + HLIC(Microprogrammed)
  - Machine cycle: 60ns, Reg.file: 40bits x 32W
  - 4 stage pipeline for RISC inst.
  - Internal Inst. Mem: 50 bits x 8 KW
  - Cache: 64 KB, 256 column, 4 sets, 32B/block
  - Protocol: Write-back, Invalidation
  - Data width: 40 bits/word
  - Shared Memory capacity: 256 MB
- Max. 512 PEs, 8 PE/cluster and 4 clusters/cabinet
- Network:
  - Double hyper-cube (Max 6 dimensions)
  - Max. 20MB/sec in each link

Figure 6: PIM model P: Main Features and Appearance of a Cabinet





- Machine language: KL1-b
- Architecture of PE:
  - Microprogram control (64 bits/word x 32 KW)
  - Data width: 40 bits/word
  - Machine cycle: 60ns, Reg.file: 40 bits x 64W
  - 5 stage pipeline
  - Cache: 1 KW for Inst., 4 KW for Data
  - Memory capacity: 16MW x 40 bits (80 MB)
- Max. 256 PEs, 32 PE/cabinet
- Network:
  - 2-dimensional mesh
  - 4.2MB/s x 2 directions/ch

Figure 7: PIM model M: Main Features and Appearance of four Cabinets

5.2.2 Development of PIMOS

PIMOS was intended to be a standard parallel operating system for large-scale parallel machines used in symbol and knowledge processing. It was designed as an independent, self-contained operating system with a programming environment suitable for KL1. Its functions for resource management and execution control of user programs were designed as independent from the architectural details of the PIM hardware. They were implemented based on an almost completely non-centralized management scheme so that the design could be applied to a parallel machine with one million element processors.[Chikayama 1992]

PIMOS is completely written in KL1. Its management and control mechanisms are implemented using a "meta-call" primitive of KL1. The KL1 language processor has embedded an automatic memory management mechanism and a dataflow synchronization mechanism. The management and control mechanisms are then implemented over these two mechanisms.

The resource management function is used to manage the memory resources and processor resources allocated to user processes and input and output devices. The program execution control function is used to start and stop user processes, control the order of execution following priorities given to them, and protect system programs from user program bugs like the usual sequential operating systems.

PIMOS supports multiple users, accesses via network and so on. It also has an efficient KL1 programming environment. This environment has some new tools for debugging parallel programs such as visualization programs which show a programmer the status of load balancing in graphical forms, and other monitoring and measurement programs.

5.2.3 Knowledge base management system

The knowledge base management system consists of two layers. The lower layer is a parallel database management system, Kappa-P. Kappa-P is a database management system based on a nested relational model. It is more flexible than the usual relational database management system in processing data of irregular sizes and structures, such as natural language dictionaries and biological databases.

The upper layer is a knowledge base management system based on a deductive object-oriented database. [Yokota and Nishio 1989] This provides us with a knowledge representation language, Quixote. [Yokota and Yasukawa 1992] These upper and lower layers are written in KL1 and are now operational on PIMOS.

The development of the database layer, Kappa, was started at the beginning of the intermediate stage.

Kappa aimed to manage the "natural databases" accumulated in society, such as natural language dictionaries. It employed a nested relational model so that it could easily handle data sets with irregular record sizes and nested structures. Kappa is suitable not only for natural language dictionaries but also for DNA databases, rule databases such as legal data, contract conditions, and other "natural databases" produced in our social systems.

The first and second versions of Kappa were developed on a PSI machine using the ESP language. The second version was completed at the end of the intermediate stage, and was called Kappa-II.[Yokota et al. 1988]

In the final stage, a parallel and distributed implementation of Kappa was begun. It is written in KL1 and is called Kappa-P. Kappa-P is intended to use large PIM main memories for implementing the main memory database scheme, and to obtain very high throughput rate for disk input and output by using many disks connected in parallel to element processors.

In conjunction with the development of Kappa-II and Kappa-P, research on a knowledge representation language and a knowledge base management system was conducted. After repeated experiments in design and implementation, a deductive object-oriented database was employed in this research.

At this point the design of the knowledge representation language, Quixote, was completed. Its language processor, which is the knowledge base management system, is under development. This language processor is being built over Kappa-P. Using Quixote, construction of a knowledge base can then be made continuously from a simple database. This will start with the accumulation of passive fact data, then gradually add active rule data, and will finally become a complete knowledge base.

The Quixote and Kappa-P system is a new knowledge base management system which has a high-level knowledge representation language and the parallel and distributed database management system as the base of the language processor. The first versions of Kappa-P and Quixote are now almost complete. It is interesting to see how this big system operates and how much its overhead will be.

5.2.4 Knowledge programming software

This software consists of various experimental programs and tools built in theoretical research and development into some element technologies for knowledge processing. Most of these programs and tools are written in KL1. These could therefore be regarded as application programs for the parallel inference system.

1. Constraint logic programming system

   In the final stage, a parallel constraint logic programming language, GDCC, is being developed. This language is a high-level logic language which has a constraint solver as a part of its language processor. The language processor is implemented in KL1 and is intended to use parallel processing to make its execution time faster. The GDCC is evaluated by experimental application programs such as a program for designing a simple handling robot.[Aiba and Hasegawa 1992]

2. Theorem proving and program transformation

   A model generation theorem prover, MGTP, is being implemented in KL1. For this application, the optimization of load balancing has been made successfully. The power of parallel processing is almost proportional to the number of element processors being used. This prover is being used as a rule-based reasoner for a legal reasoning system. It enables this system to use knowledge representation based on first order logic, and to contribute to easy knowledge programming.

3. Natural language processing

   Software tools and linguistic data bases are being developed for use in implementing natural language interfaces. The tools integrated into a library called a Language Tool Box (LTB). The LTB includes natural language parsers, a sentence generators, and the linguistic databases and dictionaries including syntactic rules and so on.

5.2.5 Benchmarking and experimental parallel application software

This software includes benchmarking programs for the parallel inference system, and experimental parallel application programs which were built for developing parallel programming methodology, knowledge representation techniques, higher-level inference mechanisms and so on.

In the final stage, we extended the application area to include larger-scale symbol and knowledge processing applications such as genetic information processing and legal expert systems. This was in addition to engineering applications such as VLSI-CAD systems and diagnostic systems for electronic equipment. [Nitta 1992]

1. VLSI CAD programs

   Several VLSI CAD programs are being developed for use in logic simulation, routing, and placement. This system is aimed at developing various parallel algorithms and load balancing methods. As there are sequential programs which have similar functions to these programs, we can compare the performance of the PIM against that of conventional machines.

2. Genetic information processing programs

   Sequence alignment programs for proteins and a protein folding simulation program are being developed. Research on an integrated database for biological data is also being made using Kappa.

3. A legal reasoning system

   This system infers possible judgments on a crime using legal rules and past cases histories. It uses the parallel theorem prover, MGTP, as a core of the rule-based reasoner. This system is making full use of important research results of this project, namely, the PIM, PIMOS, MGTP and high-level inference and knowledge representation techniques.

4. A Go game playing system

   The search space of a Go game is too large to apply any exhaustive search method. For a human player, there are many text books to show typical position sequences of putting stones which is called "Joseki" patterns. This system has some of the Joseki patterns and some heuristic rules as its knowledge base to win the game against a human player. It aims to attain 5 to 10 "kyuu" level.

The applications we have described all employ symbol and knowledge processing. The parallel programs have been programmed in KL1 in a short time. Particularly for the CAD and sequence alignment programs, the processing speed has improved almost proportionally to the number of element processors.

However, as we can see in the Go playing system, which is a very sophisticated program, the power of the parallel inference system can not always increase its intelligence effectively. This implies that we cannot effectively transcribe "natural" knowledge bases written in text books on Go into data or rules in "artificial" knowledge base of the system which would make the system " clever". We need to make more effort to find out a better program structure and better algorithms to make full use of the merit of parallel processing.

6 Evaluation of the parallel inference system

6.1 General purpose parallel programming environment

The practical problems in symbol and knowledge processing applications have been written efficiently in KL1, and solved quickly using a PIM which has several hundred element processors. Productivity of parallel software using in KL1 has been proved to be much higher than in any conventional language. This high productivity is apparently a result of using the automatic memory management mechanism and the automatic dataflow synchronization mechanism.

Our method of specifying job division and load balancing has been evaluated and proved successful. KL1 programming takes a two-step approach. In the first step, a programmer writes a program concentrating only on the program algorithms and a model. When the program is completed, the programmer adds the specifications for job division and load balancing using a notation called "pragma" as the second step. This separation makes the programming work simple and productive.

The specification of the KL1 language has been evaluated as practical and adequate for researchers. However, we realize that application programmers need a simpler and higher-level KL1 language specification which is a subset of KL1. In the future, several application-oriented KL1 language specifications should be provided, just as the von Neumann language set has a variety of different languages such as Fotran, Pascal and Cobol.

6.2 Evaluation of KL1 and PIMOS

The functions of PIMOS, some of which are implemented as KL1 functions, have been proved to be effective for running and debugging user programs on parallel hardware. The resource management and execution mechanisms in particular work as we had expected. For instance, priority control of user processes permits programmers to use about 4,000 priority levels and enables them to write various search algorithms and speculative computations very easily. We are convinced that the KL1 and PIMOS will be the best practical example for general purpose parallel operating systems in the future.

6.3 Evaluation of hardware support for language functions

In designing of the PIM hardware and the KL1 language processor, we thought it more important to provide a usable and stable platform which has a sufficient number of element processor for parallel software experiments than to build many dedicated functions into the element processor. Only the dedicated hardware support built in the element processor was tag architecture. Instead, we added more support for the interconnection between element processors such as message routing hardware and a coherent cache chip.

We did not embed complex hardware support, such as a matching store of a dataflow machine, or a content-addressable memory. We thought it risky because an implementation of the complex hardware would take a long turn around time even by a very advanced VLSI technology. We also considered that we should create a new optimization technique for a compiler dedicated to

the embedded complex hardware support, and that this would not easy too.

The completion of PIM hardware is now one year behind the original schedule, mainly because we had many unexpected problems in the design of the random logic circuits, and in submicron chip fabrication. If we had employed a more complex design for the element processor, the PIM hardware would have been further from completion.

6.3.1 Comparison of PIM hardware with commercially available technology

Rapid advances have been made in RISC processors recently. Furthermore, a few MIMD parallel machines which use a RISC processor as their element processor have started to appear in the market. When we began to design the PIM element processor, the performances of both RISC and CISC processors were as low as a few MIPS. At that time, a dedicated processor with tag architecture could attain a better performance. However, now some RISC processors have attained more than 50 MIPS. It is interesting to evaluate these RISC processors for KL1 program execution speed.

We usually compare the execution speed of a PIM element processor to that of a general-purpose microprocessor, regarding 1 LIPS as approximately equivalent to 100 IPS. This means that a 500 KLIPS PIM element processor should be comparable to a 50 MIPS microprocessor. However, the characteristics of KL1 program execution are very different from those of the usual benchmark programs for general-purpose microprocessors.

The locality of memory access patterns for practical KL1 programs is lower than for standard programs. As the length of the object codes for a RISC instruction set has to be longer than a CISC or dedicated instruction set processors, the cache miss ratio will be greater. Then, simple comparison with the PIM element processor and some recent RISC chips using announced peak performance is not meaningful. Thus, the practical implementation of the KL1 language processor on a typical RISC processor is necessary.

Most of the MIMD machines currently on the market lack a general parallel programming environment. The porting of the KL1 language processor may allow them to employ new scientific applications as well as symbol and knowledge processing applications.

In the future processor design, we believe that a general purpose microprocessor should have tag architecture support as a part of its standard functions.

6.3.2 Evaluation of high-level programming overhead

Parallel programming in KL1 is very productive, especially for large-scale and complex problems. The control of job division and load balancing works well for hundreds of element processors. No conventional language is so productive. However, if we compare the processing speed of a KL1 program with that of a conventional language program with similar functions within a single element processor, we find that the KL1 overhead is not so small. This is a common trade-off problem between high-level programming and low-level programming.

One straightforward method of compensating is to provide a simple subroutine call mechanism to link C language programs to KL1 programs. Another method is to improve the optimization techniques of compilers. This method is more elegant than the first. Further research on optimization technique should be undertaken.

7 Conclusion

It is obvious that a general-purpose parallel programming language and environment is indispensable for solving practical problems of knowledge and symbol processing. The straightforward extension of conventional von Neumann languages will not allow the use of hundreds of element processors except for regular scientific calculations.

We anticipated the difficulties in efficient implementation of the automatic memory management and synchronization mechanisms. However, this has been now achieved. The productivity and maintainability of KL1 is much higher than we expected. This more than compensates for the overhead in high-level language programming.

Several experimental parallel application programs on the parallel inference system have proved that most large-scale knowledge processing applications contain potential parallelism. However, to make full use of this parallelism, we need to have more parallel algorithms and paradigms to actually program the applications.

The research and development targets of this FGCS project have been achieved, especially as regards the parallel inference system. We plan to distribute the KL1 language processor and PIMOS as free software or public domain software, expecting that they will be ported to many MIMD machines, and will provide a research platform for future knowledge processing technology.

Acknowledgment

The development of the FGCS prototype system was conducted jointly by many people at ICOT, cooperating manufacturers, and many researchers in many countries. The author would like to express my gratitude to all the people who have given us much advise and help for more than 10 years.

References

[Uchida 1987] S. Uchida. "Inference Machines in FGCS Project", TR 278, ICOT, 1987.

[Uchida et al. 1988] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama, "Research and Development of The Parallel Inference System in The Intermediate Stage of The project", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Nov.28-Dec.2, 1988.

[Goto et al. 1988] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. " Overview of the Parallel Inference Machine Architecture (PIM)", In Proc. of the International Conference on Fifth Generation Computing Systems 1988, Tokyo, Japan, November 1988.

[Taki 1992] K. Taki, "Parallel Inference Machine, PIM", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Jul.1-5, 1992.

[Chikayama 1984] T. Chikayama, "Unique Features of ESP", In Proc. Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, 1984, pp. 292-298.

[Warren 1983] D.H.D. Warren, "An Abstract Prolog Instruction Set", Technical Note 309, Artificial Intelligence Center, SRI, 1983.

[Clark adn Gregory 1983] Keith L. Clark and Steve Gregory, "Parlog: A parallel logic programming language", Research Report TR-83-5, Imperial College, March 1983.

[Clark and Gregory 1984] K. L. Clark and S. Gregory, "Notes on Systems Programming in PARLOG", In Proc. Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, 1984, pp. 299-306.

[Shapiro 1983] E. Y. Shapiro, "A subset of Concurrent Prolog and Its Interpreter", TR 003, ICOT, 1987.

[Ueda 1986] K. Ueda. Guarded Horn Clauses, "In Logic Programming", '85, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, 1986, pp.168-179.

[Ueda 1986] K. Ueda, "Introduction to Guarded Horn Clauses", TR 209, ICOT, 1986.

[Chikayama and Kimura 1985] T. Chikayama and Y. Kimura, "Multiple Reference Management in Flat GHC", In Proc. Fourth Int. Conf. on Logic Programming, MIT Press, 1987, pp. 276-293.

[Chikayama el al. 1988] T. Chikayama, H. Sato and T. Miyazaki, "Overview of the Parallel Inference Machine Operating System (PIMOS)", In Proc. Int. Conf. on Fifth Generation Computer Systems 1988, ICOT, 1988, pp. 230-251.

[Chikayama 1992] T. Chikayama, "Operating System PIMOS and Kernel Language KL1", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Jul.1-5, 1992.

[Uchida et al. 1988] S. Uchida,"The Research and Development of Natural Language Processing Systems in the Intermediate Stage of the FGCS Project", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Nov.28-Dec.2, 1988.

[Yokota et al. 1988] K. Yokota, M. Kawamura, and A. Kanaegami, "Overview of the Knowledge Base Management System (KAPPA)", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Nov.28-Dec.2, 1988.

[Yokota and Nishio 1989] K. Yokota and S. Nishio, "Towards Integration of Deductive Databases and Object-Oriented Databases–A Limited Survey", Proc. Advanced Database System Symposium, Kyoto, Dec., 1989.

[Yokota and Yasukawa 1992] K.Yokota and H. Yasukawa, "Towards an Integrated Knowledge-Base Management System", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Jul.1-5, 1992.

[Aiba and Hasegawa 1992] A. Aiba and R. Hasegawa, "Constraint Logic Programming System", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Jul.1-5, 1992.

[Nitta 1992] K. Nitta, K. Taki, and N. Ichiyoshi, "Development of Parallel Application Programs of the Parallel Inference Machine", Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Jul.1-5, 1992.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Inference Machine PIM

Kazuo Taki

First Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, JAPAN
taki@icot.or.jp

Abstract

The parallel inference machine, PIM, is the prototype hardware system in the Fifth Generation Computer Systems (FGCS) project. The PIM system aims at establishing the basic technologies for large-scale parallel machine architecture, efficient kernel language implementation and many aspects of parallel software, that must be required for high performance knowledge information processing in the 21st century. The PIM system also supports an R & D environment for parallel software, which must extract the full power of the PIM hardware.

The parallel inference machine PIM is a large-scale parallel machine with a distributed memory structure. The PIM is designed to execute a concurrent logic programming language very efficiently. The features of the concurrent logic language, its implementation, and the machine architecture are suitable not only for knowledge processing, but also for more general large problems that arise dynamic and non-uniform computation. Those problems have not been covered by commercial parallel machines and their software systems targeting scientific computation. The PIM system focuses on this new domain of parallel processing.

There are two purposes to this paper. One is to report an overview of the research and development of the PIM hardware and its language system. The other is to clarify and itemize the features and advantages of the language, its implementation and the hardware structure with the view that the features are strong and indispensable for efficient parallel processing of large problems with dynamic and non-uniform computation.

1 Introduction

The Fifth Generation Computer Systems (FGCS) project aims at establishing basic software and hardware technologies that will be needed for high-performance knowledge information processing in the 21st century. The parallel inference machine PIM is the prototype hardware system and offers gigantic computation power to the knowledge information processing. The PIM system includes an efficient language implementation of KL1, which is the kernel language and a unique interface between hardware and software.

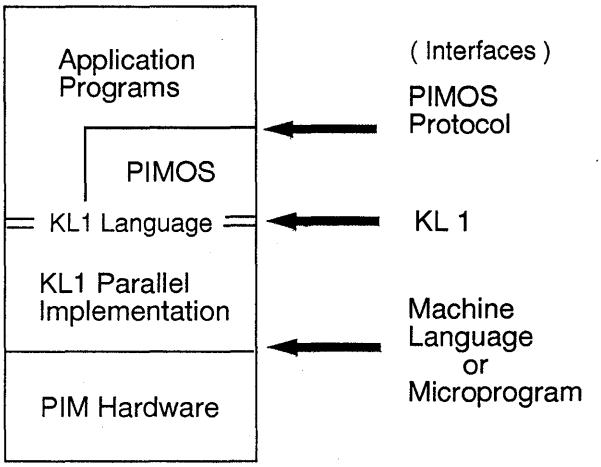


Figure 1: Overview of the PIM System

Logic programming was chosen as the common basis of research and development for the project. The primary working hypothesis was as follows. "Many problems of future computing, such as execution efficiency (of parallel processing), descriptive power of languages, software productivity, etc., will be solved drammatically with the total reconstruction of those technologies based on *logic programming*.

Following the working hypothesis, R & D on the PIM system started from scratch with the construction of hardware, a system software, a language system, application software and programming paradigms, all based on *logic programming*. Figure 1 gives an overview of the system structure.

The kernel language KL1 was firstly designed for efficient concurrent programming and parallel execution of knowledge processing problems. Then, R & D on the PIM hardware with distributed-memory MIMD architecture and the KL1 language implementation on it were carried out, both aiming at efficient KL1 execution in

parallel. A machine roughly with 1000 processors was primarily targeted. Each of these processors was to be a high-speed processor with hardware support for symbolic processing. The PIM system also focused on realizing a useful R & D environment for parallel software which could extract the real computing power of the PIM. The preparation of a good R & D environment was an important project policy.

KL1 is a concurrent logic programming language primarily targeting knowledge processing. Since the language had to be a common basis for various types of knowledge processing, it became a general-purpose concurrent language suitable for symbolic processing, without shifting to a specific reasoning mechanism or a certain knowledge representation paradigm.

Our R & D led to the language features of KL1 being very suitable for covering *the dynamic and non-uniform large problems* that are not covered by commercial parallel computers and their software systems for scientific computation. Most knowledge processing problems are included in the problem domain of *dynamic and non-uniform computation.* The PIM hardware and the KL1 language implementation support the efficiency of the language features. Thus, the PIM system covers this new domain of parallel processing.

This paper focuses on two subjects. One is the R & D report of the PIM hardware and the KL1 language implementation on it. The other is to clarify and itemize the features and advantages of the language, its implementation and the hardware structure with the view that the features are strong and indispensable for efficient parallel processing of large problems with *dynamic and non-uniform computation.* Any parallel processing system targeting this problem domain must consider those features.

Section 2 scans the R & D history of parallel processing systems in the FGCS project, with explanation of some of the keywords. Section 3 characterizes the PIM system. Many advantageous features of the language, its parallel implementation and hardware structure are described with the view that the features are strong and indispensable for efficient programming and execution of *the dynamic and non-uniform large problems.* Section 4 presents the machine architecture of PIM. Five different models have been developed for both research use and actual software development. Some hardware specifications are also reported. Section 5 briefly describes the language implementation methods and techniques, to give a concrete image of several key features of the KL1 implementation. Section 6 reports some measurements and evaluation mainly focusing on a low-cost implementation of small-grain concurrent processes and remote synchronization, which support the advantageous features of KL1. Overall efficiency, as demonstrated by a few benchmark programs, is shown, including the most recent measurements on PIM/m. Then, section 7 concludes this paper.

Several important research issues of parallel software are reported in other papers: the parallel operating system PIMOS is reported in [Chikayama 1992] and the load balancing techniques controlled by software are reported in [Nitta *et al.* 1992].

2 R & D History

This section shows the R & D history of parallel processing systems in the FGCS project. Important research items and products of the R & D are described briefly, with explanations of several keywords. There are related reports for further information [Uchida 1992] [Uchida *et al.* 1988].

2.1 Start of the Mainstream of R & D

Mainstream of R & D of the parallel processing systems started at the beginning of the intermediate stage of the FGCS project, in 1985. Just before that time, a concurrent logic language GHC [Ueda 1986] had been designed, which was chosen as the kernel language of the R & D. Language features will be described in section 3.4.

Development of small hardware and software systems was started based on the kernel language GHC as a hardware and software interface. The hardware system was used as a testbed of parallel software research. Experiences and evaluation results was fed back to the next R & D of larger hardware and software system, which was the *bootstrapping of R & D.*

It was started from development of the Multi-PSI [Taki 1988]. Purpose of the hardware development was not only the architectural research of a knowledge processing hardware, but also a preparation of a testbed for efficient language implementation of the kernel language. The Multi-PSI also focused to be a useful tool and environment of parallel software research and development. That is, the hardware was not just an experimental machine, but a reliable system being developed in short period, with measurements and debugging facilities for software development. After construction of the Multi-PSI/V1 and /V2 with language implementations, various parallel programs and technology and knowhow of parallel software have been accumulated [Nitta *et al.* 1992] [Chikayama 1992]. The systems have been used for the advanced software development environment for the parallel inference machines.

2.2 Multi-PSI/V1

The first hardware was the Multi-PSI/V1 [Taki 1988] [Masuda *et al.* 1988], started in operation in spring 1986. The personal sequential inference machine PSI [Taki *et al.* 1984] was used for processing elements. It was a development result of the initial stage of the

project. Six PSI machines were connected by a mesh network, which supported so called *wormhole routing*. The first distributed implementation of GHC was built on it [Ichiyoshi *et al.* 1987]. (Distributed implementation means a parallel implementation on a distributed memory hardware). Execution speed was slow (1K LIPS = logical inference per second) because an interpreter system was written in ESP (the system description language of the PSI). However, basic algorithms and techniques of distributed implementation of GHC was investigated in it. Several small parallel programs were written and executed on it for evaluation, and primary experimentations of load balancing were also carried out.

2.3 From GHC To KL1

Since GHC had only basic functions that the kernel concurrent logic language had to support, language extensions were needed for the next more practical system. Kernel language KL1 was designed with considerations of execution efficiency, operating system supports, and some built-in functions [Ueda and Chikayama 1990] [Chikayama 1992]. An intermediate language KL1-B, which was the target language of KL1 compiler, was also designed [Kimura and Chikayama 1987]. In the Multi-PSI/V2 and a PIM model, binary code of KL1-B is directly interpreted by microprogram; that is, KL1-B is machine language itself. In the other PIM models, KL1-B code is converted to lower-level machine instruction sequences and executed by hardware.

2.4 Multi-PSI/V2

The second hardware system was the Multi-PSI/V2 [Takeda *et al.* 1988] [Nakajima 1992], which was improved in performance and functions enough to be called as the first experimental parallel inference machine. It started in operation in 1988 and was demonstrated in the FGCS'88 international conference.

The Multi-PSI/V2 included 64 processors, each of which were equivalent to the CPU of PSI-II [Nakashima and Nakajima 1987], smaller and faster model of the PSI. Processors were connected with two dimensional mesh network with improved speed (10M Bytes/s, full duplex in each channel). KL1-B was the machine language of the system, executed by microprogram. Almost all the runtime functions of KL1 was implemented in microprogram. The KL1 implementation was improved much in execution efficiency, reducing inter-processor communication messages, efficient garbage collections, etc. compared with Multi-PSI/V1. It attained 130K LIPS (in KL1 append) in single processor speed. Table 1 to 4 include specifications of the Multi-PSI/V2. Since 1988, more than 15 systems, large system with 64 processors and small with 32 or 16 processors, have been in operation for parallel software R & D in ICOT and in cooperating companies.

A strong simulator of the Multi-PSI/V2 was also developed for software development environment. It was called the pseudo Multi-PSI, available on the Prolog workstation, PSI-II. A very special feature was caused by similarity of the PSI-II CPU and processing element of the Multi-PSI/V2. Usually, PSI-II executed ESP language with dedicated microprogram. However, it loaded KL1 microprogram dynamically at the activation of the simulator system. The simulator executed KL1 programs as similar speed as that of the Multi-PSI/V2 single processor. Since the PIMOS could be also executed on the simulator, programmers could use the simulator as similar environment as the real Multi-PSI/V2, except for speedup with multiple processors and process scheduling. The pseudo Multi-PSI was the valuable system for initial debugging of KL1 programs.

2.5 Software Development on the Multi-PSI/V2

Parallel operating system PIMOS (the first version) and four small application programs (benchmark programs) [Ichiyoshi 1989] had been developed until FGCS'88. Much efforts was paid in PIMOS development to realize a good environment of programming, debugging, execution and measurements of parallel programs. In the development of small application programs, several important research topics of parallel software were investigated, such as concurrent algorithms with large concurrency without increase of complexity, programming paradigms and techniques of efficient KL1 programs, and dynamic and static load balancing schemes for dynamic and non-uniform computation.

The PIMOS has been improved in several versions, and ported to the PIM until 1992. The small application programs, pentomino [Furuichi *et al.* 1990], best-path [Wada and Ichiyoshi 1990], PAX (natural language parser) and tsume-go (a board game) were improved, measured and analyzed until 1989. They are still used as test and benchmark programs on the PIM.

These development gave observations that the KL1 system on the Multi-PSI/V2 with PIMOS has reached sufficient performance level for practical usage, and has realized sufficient functions for describing complex concurrent programs and for experimentations of software-controlled load balancing.

Several large-scale parallel application programs have been developed from late 1989 [Nitta *et al.* 1992] and still continuing. Some of them have been ported to the PIM.

2.6 Parallel Inference Machine PIM

2.6.1 Five PIM Models

Design of the parallel inference machine PIM was started in concurrent with manufacturing of the Multi-PSI/V2. Some research items in hardware architecture were omitted in the development of the Multi-PSI/V2, because of short development time needed for starting the parallel software development. So, PIM took a greedy R & D plan, focusing both the architectural research and realization of software development environment.

The first trial to the novel architecture was the multiple clusters. A small number of tightly-coupled processors with shared-memory formed a cluster. Many clusters were connected with high speed network to construct the PIM system with several hundred processors. Benefits of the architecture will be discussed in section 3.7.

Many component technologies had to be developed or improved to realize the new system, such as parallel cache memory suitable for frequent inter-processor communications, high speed processors for symbolic processing, improvement of the network, etc. For R & D of better component technologies and their combinations, the development plan of five PIM models was made, so that different component architecture and their combinations could be investigated with assigning independent research topics or roll on each model.

Two models, PIM/p [Kumon *et al.* 1992] and PIM/c [Nakagawa *et al.* 1992], took the multi-cluster structure. They include several hundreds processors, maximum 512 in PIM/p and 256 in PIM/c. They were developed both for the architectural research and software R & D. Each investigated different network architecture and processor structure.

The other two models, PIM/k [Sakai *et al.* 1991] and PIM/i [Sato *et al.* 1992], were developed for the experimental use of intra-cluster architecture. Two-layered coherent cache memory which enabled larger number of processors in a cluster, broadcast-typed coherent cache memory, and a processor with LIW-type instruction set were tested.

The other model, PIM/m [Nakashima *et al.* 1992], did not take the multi-cluster structure, but focused the rigid compatibility with the Multi-PSI/V2, having improved processor speed and larger number of processors. The maximum number of processors will be 256. The performance of a processor will be four to five times larger at peek speed, and 1.5 to 2.5 times larger in average than the Multi-PSI/V2. The processor was similar to the CPU of PSI-UX, the most recent version of the PSI machine. A simulator, pseudo-PIM/m, was also prepared like the pseudo Multi-PSI. The PIM/m targeted the parallel software development machine mostly among the models.

Architecture and specifications of each model will be reported in section 4.

Experimental implementations of some LSIs of these models have started in 1989. The final design was almost fixed in 1990, and manufacturing of whole system was proceeded with in 1991. From 1991 to spring 1992, assembly and test of the five models have carried on.

2.6.2 Software Compatibility

KL1 language is common among all the five PIM models. Except for execution efficiency, any KL1 programs including PIMOS can run on the all models. Hardware architecture is different between two groups, Multi-PSI and PIM/m as the one, and the other PIM models as the other. However, from programmers' view, abstract architecture are designed similar as follows.

The load allocation to processors are fully controlled by programs on the Multi-PSI and the PIM/m. It is sometimes written by programmers directly, and sometimes specified by load allocation libraries. Programmers are often researchers of load balancing techniques. On the other hand, load balancing in a cluster is completely controlled by the KL1 runtime system (not by KL1 programs) among the PIM models with the multi-cluster structure. That is, programmers does not have to think of multiple processors in a cluster, but specify load allocation to each cluster in their programs. It means that a processor of the Multi-PSI or PIM/m corresponds to a cluster of the PIM models with the multi-cluster structure, which simplifies portation of KL1 programs.

2.7 KL1 Implementation for PIM

KL1 system must be the first regular system in the world which can execute large-scale parallel symbolic processing programs very efficiently. Execution mechanisms or algorithms of KL1 language had been developed for distributed memory architectures sufficiently on the Multi-PSI/V2. Some mechanisms and algorithms should be expanded for the multi-cluster architecture of PIM. Ease of porting the KL1 system to four different PIM models was also considered in the language implementation method. Only the PIM/m inherited the KL1 implementation method directly from the Multi-PSI/V2.

To expand the execution mechanisms or algorithms suitable for the multi-cluster architecture, several technical topics were focused, such as avoiding data update contentions among processors in a cluster, automatic load balancing in a cluster, expansion of an inter-cluster message protocol applicable for the message outstripping, parallel garbage collection in a cluster, etc. [Hirata *et al.* 1992].

For easiness of porting the KL1 system to four different PIM models, a common specification of KL1 system "VPIM (virtual PIM)" was written in "C"-like description language "PSL", targeting a common virtual hardware. VPIM was the executable specification of KL1 execution algorithms, which was translated to C language and executed to examine the algorithms. VPIM has been

translated to lower-level machine languages or microprograms automatically or by hands according to each PIM structure.

Preparation of the description language started in 1988. Study of efficient execution mechanisms and algorithms continued until 1991, then, VPIM was completed. Porting the VPIM to four PIM models partially started in autumn 1990, and continued to spring 1992. Now, the KL1 system with PIMOS is available on each PIM model. On the other hand, KL1 system on the PIM/m, which was implemented in microprogram, was made from conversion of Multi-PSI/V2 microprogram by hands or partially in automatic translation. Prior to the other PIM models, PIM/m started in operation with the KL1 system and PIMOS in summer 1991.

2.8 Performance and System Evaluation

Measurements, analysis, and evaluation should be done on various levels of the system shown below.

1. Hardware architecture and implementations
2. Execution mechanisms or algorithms of KL1 implementation
3. Concurrent algorithms of applications (algorithms for problem solving, independent from mapping) and their implementations
4. Mapping (load allocation) algorithms
5. Total system performance of a certain application program on a certain system

Various works have been done on the Multi-PSI/V2. 1 and 2 were reported in [Masuda *et al.* 1988] and [Nakajima 1992]. 3 to 5 were reported in [Nitta *et al.* 1992], [Furuichi *et al.* 1990], [Ichiyoshi 1989] and [Wada and Ichiyoshi 1990].

Primary measurements have just started on each PIM models. Some intermediate results are included in [Nakashima *et al.* 1992] and [Kumon *et al.* 1992].

Total evaluation of the PIM system will be done in the near future, however, some observations and discussions are included in section 6.

3 Characterizing the PIM and KL1 system

PIM and KL1 system have many advantageous features for very efficient parallel execution of large-scale knowledge processing which often shows very dynamic runtime characteristics and non-uniform computation, much different from numerical applications on vector processors and SIMD machines.

This section clarifies the characteristics of the targeted problem domain shortly, and describes the various advantageous features of PIM and KL1 system, that are dedicated for the efficient programming and processing in the problem domain. They will give the total system image and help to clarify the difference and similarity of the system with other large-scale multiprocessors, recently available in the market.

3.1 Summary of Features

The total image of PIM and KL1 system are briefly scanned as follows. Detailed features and their benefits, and reasons why they were chosen are presented in the following sections.

Distributed memory MIMD machine:
Global structure of the PIM is the distributed memory MIMD machine in which hundreds computation nodes are connected by highspeed network. Scalability and ease of implementations are focused. Each computation node includes single processor or several tightly-coupled processors, and large memory. Processors are dedicated for efficient symbolic processing.

Logic programming language: The kernel language KL1 is a concurrent logic programming language, which is single language for system and application descriptions. Language implementation and hardware design are based on the language specification.

KL1 is not a high-level knowledge representation language nor a language for certain type of reasoning, but a general-purpose language for concurrent and parallel programming, especially suitable for symbolic computations.

KL1 has many beneficial features to write parallel programs in those application domains, described below.

Application domain: Primary applications are large-scale knowledge processing and symbolic computation. However, large numerical computation with dynamic features, or with non-uniform data and non-uniform computation (non-data-parallel computation) are also targeted.

Language implementation: One KL1 system is implemented on a distributed memory hardware, which is not a collection of many KL1 systems implemented on each processing node. A global name space is supported for code, logical variables, etc. Communication messages between computation nodes are handled implicitly in KL1 system, not by KL1 programs. An efficient implementation for small-grain concurrent processes is taken.

These implementations focus to realize the beneficial features of KL1 language for the application domains described before.

Policy of load balancing: Load balancing between computation nodes should be controlled by KL1 programs, not by hardware nor by the language system automatically. Language system has to support enough functions and efficiency for the experiments of various loadbalancing schemes with software.

3.2 Basic Choices

(1) **Logic programming:** The first choice was to adopt logic programming as the basis of the kernel language. The decision is mainly due to the insights of ICOT founders, who expected that logic programming was suitable for both knowledge processing and parallel processing. A history, from vague expectations on logic programming to the concrete design of the KL1 language, is explained in [Chikayama 1992].

(2) **Middle-out approach:** A middle-out approach of R & D was taken, placing the KL1 language as the central layer. Based on the language specification, design of the hardware and the language implementation started downward, and writing the PIMOS operating system and parallel software started upward.

(3) **MIMD machine:** The other choices concerned with basic hardware architecture.

Dataflow architecture before mid 1980 was considered not providing enough performance against hardware costs, according to observations for research results in initial stage of the project.

SIMD architecture seemed inefficient on applications with dynamic characteristics or low data-parallelism that are often seen in knowledge processing.

MIMD architecture remained without major demerits and was most attractive from the viewpoint of ease of implementation with standard components.

(4) **Distributed memory structure:** Distributed memory structure is suitable to construct very large system, and easy to implement.

Recent large-scale shared memory machines with directory-based cache coherency mechanisms claims good scalability. However, when the block size (the coherency management unit) is large, the interprocessor communication with frequent small data transfer seems inefficient. KL1 programs require the frequent small data transfer. When the block size becomes small, large directory memory is needed, which increases the hardware cost.

Single assignment languages need special memory management such as dynamic memory allocation and garbage collection. These management should be done as locally as possible for the sake of efficiency. Local garbage collection requires separation of local and global address spaces with some indirect referencing mechanism or address translation, even in a scalable shared memory architecture. Merits of the low-cost communication in the shared memory architecture decrease significantly for such the case.

These are the reasons to choose the distributed memory structure.

3.3 Characterizing the Applications

(1) **Characterization:** Characteristics of knowledge processing and symbolic computation are often much different from those of numerical computation on vector processors and SIMD machines. Problem formalizations for those machines usually based on data-parallelism, parallelism for regular computation on uniform data.

However, the characteristics of knowledge and symbolic computations on parallel machines tend to be very dynamic and non-uniform. Contents and amount of computation vary dynamically depending on time and space. For example, when a heuristic search problem is mapped on a parallel machine, workload of each computation node changes drastically depending on expansion and pruning of the search tree. Also, when a knowledge processing system is constructed from many heterogeneous objects, each object arises non-uniform computation. Computation loads of these problems are hardly estimated before execution.

Some classes of large numerical computation without data-parallelism also show the dynamic and non-uniform characteristics.

Those problems which has dynamism and non-uniformity of computation are called *the dynamic and non-uniform problems* in this paper, implying not only the knowledge processing and symbolic computation but also the large numerical computation without data-parallelism.

The dynamic and non-uniform problems tends to include the programs with more complex program structure than the data-parallel problems.

(2) **Requirements for the system:** Most of the software systems on recent commercial MIMD machines with hundreds of processors target the data-parallel computation, but they almost don't care other paradigms.

The dynamic and non-uniform problems arise new requirements mainly on software systems and a few on hardware systems, which are listed below.

1. Descriptive power for complex concurrent programs
2. Easy to remove bugs
3. Ease of dynamic load balancing
4. Flexibility for changing the load allocation and scheduling schemes to cope with difficulty on estimating actual computation loads before execution

3.4 Characterizing the Language

This subsection itemizes several advantageous features of KL1 that satisfy the requirements listed in the previous section. Features and characteristics of the concurrent logic programming language KL1 are described in detail in [Chikayama 1992].

The first three features have been in GHC, the basic specifications of KL1. These features make descriptive power of the language large enough to write complex concurrent programs. They are the features of *concurrent programming* to describe logical concurrency, independent from mapping to actual processors.

(1) **Dataflow synchronization:** Communication and synchronization between KL1 processes are performed implicitly at all within a framework of usual unification. It is based on the dataflow model. Implicitness is available even in a remote synchronization. The feature drastically reduces bugs of synchronization and communication compared with the case of explicit description using separate primitives. The single-assignment property of logic variables supports the feature.

(2) **Small-grain concurrent processes:** The unit of concurrent execution in KL1 is each body goal of clauses, which can be regarded as a process invocation. KL1 programs can thus involve a large amount of concurrency implicitly.

(3) **Indeterminacy:** A goal (or process) can test and wait for the instantiation of multiple variables concurrently. The first instantiation resumes the goal execution, and when a clause is committed (selected from clauses that succeed to execute guard goals), the other wait conditions are thrown away. This function is valuable to describe "non-rigid" processing within a framework of side-effect free language. Speculative computation can be dealt with, and dynamic load distribution can be also written.

The next features have been included in KL1 as extensions to GHC. (4) was introduced to describe mapping (load allocation) and scheduling. They are the features for *parallel programming* to control actual parallelism among processing nodes. (5) is prepared for operating system supports. (6) is for the efficiency of practical programs.

(4) **Pragma:** Pragma is a notation to specify goal allocation to processing nodes or specify execution priority of goals. Pragma doesn't affect the semantics of a program, but controls parallelism and efficiency of actual parallel execution. Pragmas are usually attached to goals after making sure that the program is correct anyway. It can be changed very easily because it is syntactically separated from the correctness aspect of a program.

Pragma for load allocation: Goal allocation is specified with a pragma, @node(X). X can be calculated in programs. Coupled with (1) and (2), the load allocation pragma can realize very flexible load allocation. Also coupled with (3) and the pragma, KL1 can describe a dynamic load balancing program within a framework of the pure logic programming language without side-effect. Dynamic load balancing programs are hard to be written in pure functional languages without indeterminacy.

Pragma for execution priority: Execution priority is specified with a pragma, @priority(Y). More than thousands priority levels are supported to control goal scheduling in detail, without rigid ordering.

Combination of (3) and the priority pragma realizes the efficient control of speculative computations. Large number of priority levels can be utilized in e.g. parallel heuristic search to expand good branch of the search tree at first.

(5) **Shoen function (meta-control for goal group) :** The *shoen* function is designed to handle a set of goals as a task, a unit of execution and resource management. It is mainly used in PIMOS. Start, stop and abortion of tasks can be controlled. Limit of resource consumption can be specified. When errors or exception conditions occur, the status are frozen and reported outside the *shoen*.

(6) **Functions for efficiency:** KL1 has several built-in functions or data types whose semantics is understood within the framework of GHC but which has been provided for the sake of efficiency. Those functions hide demerits of side-effect free languages, and also avoid an increase of computational complexity compared with sequential programs.

3.5 Characterizing the Language Implementation

Language features, just described in the previous section, satisfy the requirements for a system by *the dynamic and non-uniform problems* discussed in section 3.3. Most of special features of the language implementation focused to enlarge those advantageous features of KL1 language.

(1) Implicit communication:
Communication and synchronization among concurrent processes are implicitly done by unifications on shared logical variables. They are supported both in a computation node and between nodes. It is especially beneficial that a remote synchronization is done implicitly as well as local.

A process (goal) can migrate between computation nodes only being attached a pragma, `@node(X)`. When the process has reference pointers, remote references are generated implicitly between the computation nodes. The remote references are used for the remote synchronizations or communications.

These functions hide the distributed memory hardware from the "concurrent programming". That is, programmers can design concurrent processes and their communications, independent from their allocations to a same computation node or different nodes. Only the "parallel programming" with pragmas, a design of load allocation and scheduling, has to concern with hardware structure and network topology.

Implementation features of those functions are summarized below, including the features for efficiency.

- Global name space on a distributed memory hardware — in which implicit pointer management among computation nodes are supported for logical variables, structured data and program code
- Implicit data transfer caused by unifications and goal (process) migration
- Implicit message sending and receiving invoked with data transfer and goal sending, including message composition and decomposition
- Message protocols able to reduce the number of messages, and also protocols applicable to message outstripping

(2) Small-grain concurrent processes: Efficient implementation of small-grain concurrent processes are realized, coupled with low-cost communications and synchronizations among them.

Process scheduling with low-cost suspension and resumption, and priority management are supported.

Efficient implementation allows actual use of a lot of small-grain processes to realize large concurrency. A large number of processes also gives flexibility for the mapping and load balancing.

Automatic load balancing in a cluster is also supported. It is a process (goal) scheduling function in a cluster implemented with priority management. The feature hides multiprocessors in a cluster from programmers. They do not have to think about load allocation in a cluster, but only have to prepare enough concurrency.

(3) Memory management: These garbage collection mechanisms are supported.

- Combination of incremental garbage collection with subset of reference counting and stop-and-collect copying garbage collection
- Incremental releasing of remote reference pointers between computation nodes with weighted reference counting scheme

Dynamic memory management including garbage collections looks essential both for symbolic processing and for parallel processing of *the dynamic and non-uniform problems*. Because the single assignment feature, strongly needed for the problems, requires dynamic memory allocation and reclamation.

Efficiency of garbage collectors is one of key features for practical language system of parallel symbolic processing.

(4) Implementation of shoen function: *Shoen* represents a group of goals (processes) as presented in the previous subsection. *Shoen* mechanism is implemented not only in a computation node but also among nodes. Namely, processes in a task can be distributed among computation nodes, and still controlled all together with *shoen* functions.

(5) Built-in functions for efficiency: Several built-in functions and data types are implemented to keep up with the efficiency of sequential languages.

(6) Including OS kernel functions: Figure 2 shows the relation of KL1 implementation and operating system functions. KL1 implementation includes so called OS kernel functions such as memory management, process management and scheduling, communication and synchronization, virtual single name space, message composition and decomposition, etc. While, PIMOS includes upper OS functions like programming environment and user interface.

The reason why the OS kernel functions are included in the KL1 implementation is that the implementation needs to use those functions with as light cost as possible. Cost of those functions affect the actual

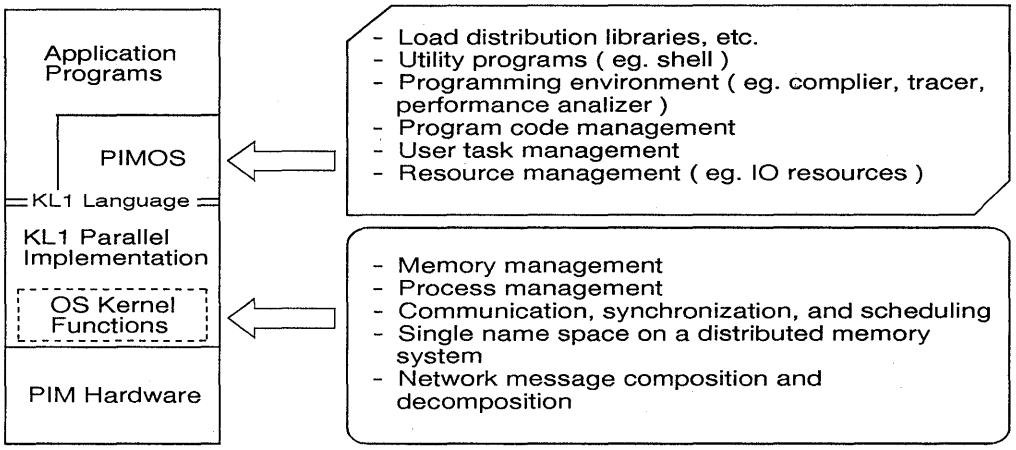


Figure 2: KL1 Implementation and OS Functions

execution efficiency of the advantageous features of KL1 language, such as large number of small-grain concurrent processes, implicit synchronization and communication among them (even between remote processes), indeterminacy, scheduling control with large number of priority levels, process migration specified with pragmas, etc. Those features are indispensable for concurrent and parallel programming and efficient parallel execution of large-scale symbolic computation with dynamic characteristics, or large-scale non-data-parallel numerical computations.

Considering a construction of similar purpose parallel processing system on a standard operating system, interface level to the OS kernel may be too high (or may arise too much overhead). Some reconstruction of OS implementation layers might be needed for the standard parallel operating systems for those large-scale computation with dynamic characteristics.

3.6 Policy of Load Balancing

Such a basic policy has been taken that load balancing between computation nodes should be completely controlled by KL1 programs, not by hardware nor by language system automatically. There are two reasons.

One is that KL1 can describe load balancing programs within usual logic programming features. Since many research topics on load distribution have been remained unsolved especially on dynamic problems, experiments on software controlled load balancing is advantageous in an aspect of flexibility. It does not include significant overhead because the KL1 language system realize a very low-cost implementation.

The other is that distributed memory architecture needs strong locality of computation, for which some programmers' help is important for better load balancing.

Language system has to support enough functions and efficiency for the experiments of various load balancing schemes by software.

Some load balancing schemes are prepared as utility programs, available for application programmers.

3.7 Characterizing the Hardware Architecture

Features of PIM hardware architecture are listed below. Some of them are specialized for symbolic processing and large-scale parallel computation of dynamic problems, and some of them are standard.

(1) Distributed memory MIMD machine: Target hardware is the large-scale MIMD machine with distributed memory structure. Hundreds processing nodes are connected by highspeed network. It was a basic choice of the R & D. The structure was considered to have large scalability, to be mostly easy for implementation, and to be suitable to separate local garbage collections and global.

(2) Cluster structure: Eight processors, that are tightly coupled with shared bus and shared memory, form a cluster. Many clusters are connected with highspeed network to form the total system. Programmers deal with a cluster as a computation node with large computation power and large memory, since automatic load balancing is supported by language system within a cluster.

Cluster is a substructure of the PIM, realizing a low latency and high bandwidth connection between processors. There are two major advantages of

the cluster structure. The first is its applicability to those problems which have less locality, while distributed memory architecture hardly processes those problems efficiently. The second is higher efficiency of memory usage compared with full distributed memory systems with the same memory size. A substructure with higher bandwidth inter-processor connection is effective to reduce needs of memory size per processor, keeping the same efficiency of parallel processing. It affects the total system cost significantly.

A disadvantage is heterogeneous inter-processor connections that increase the complexity of hardware implementations, however, the cluster with tightly coupled processors will be a standard component in the near future.

(3) **Large memory against processing power:** Non-uniform computation or dynamic computation with wide variation of grain size require larger memory to keep the processing efficiency, compared with data-parallel computation. Because extra work is needed to fill the idling time caused by irregular synchronization, which requires more working space in a memory.

(4) **Highspeed network:** Highspeed network connection between processing nodes has already become standard. However, the ratio of network load and processor load, caused by network communications, is different from the case of numerical processing. Management of virtual single name space usually arises extra processor loads for each communications, compared with the case of simple data transfer in numerical processing. It causes less needs to network bandwidth against processing power.

On the other hand, parallel symbolic computation with dynamic features often arises remote synchronizations with small data transfer. Response of the network communication is more important than bandwidth for such cases.

(5) **Coherent cache memory:** Each processor in a cluster has coherent cache memory with write back strategy. Basic technology is similar to the standard coherent cache memory used in commercial tightly coupled multiprocessors. However, the occurrence of cache to cache data transfer, caused by inter-processor communications, is larger than the usual time sharing use of commercial multiprocessors. Optimizations of cache commands and bus protocols for such usage is important to reduce bus traffic.

(6) **Dedicated processors:** Processors include special features of tag handling, data type checking and branching, and dereferencing pointers for efficient KL1 execution. These features are useful not only for symbolic processing, but also for an efficient implementation of a single-assignment language needed for the parallel processing of *the dynamic and non-uniform problems.*

The processors have dedicated instruction sets derived from the abstract instruction set KL1-B.

Pipelining and RISC-like instruction sets are also used, that are standard techniques.

4 Machine Architecture and Hardware

Overall structure and features of the PIM system were presented in the previous section. This section shows the machine architecture, hardware implementations and some technical data of each PIM models in detail.

4.1 Overview of Five PIM Models

Five PIM models have been developed, that have different architectures or different combinations of component technologies, and have different rolls of R & D.

PIM/p : PIM/p is the largest PIM model which contains maximum 512 processors. PIM/p focuses both architectural research and actual use in software R & D.

PIM/p took the multi-cluster architecture shown in Figure 3. Maximum 64 clusters can be connected. Connection network took hypercube topology. Two independent networks are connected to each clusters.

Each cluster contains eight processors connected with a shared bus and shared memory. A processor has coherent cache memory, a network interface unit "NIU", and an I/O device interface (SCSI bus) [Kumon *et al.* 1992].

Processors in all PIM models have SCSI buses, which are used to connect FEPs (Front End Processors) and hard disks. The PSI-UX [Nakashima *et al.* 1992] is used for the FEP, as an intelligent I/O device for human-machine interface.

PIM/m : PIM/m targets the software development machine and rigid compatibility with the Multi-PSI/V2. 256 processors are connected with two dimensional mesh network. The structure is shown in Figure 4. 32 hard disks, which are 20GB in total, and many FEPs are connected [Nakashima *et al.* 1992].

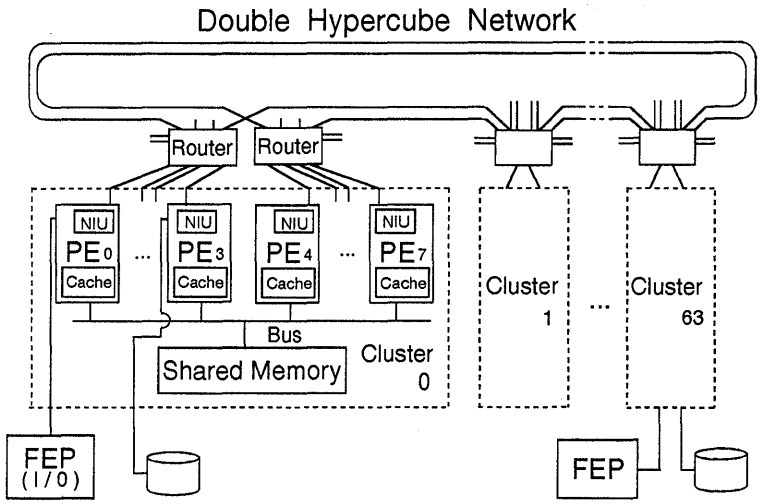


Figure 3: Overview of PIM/p Architecture

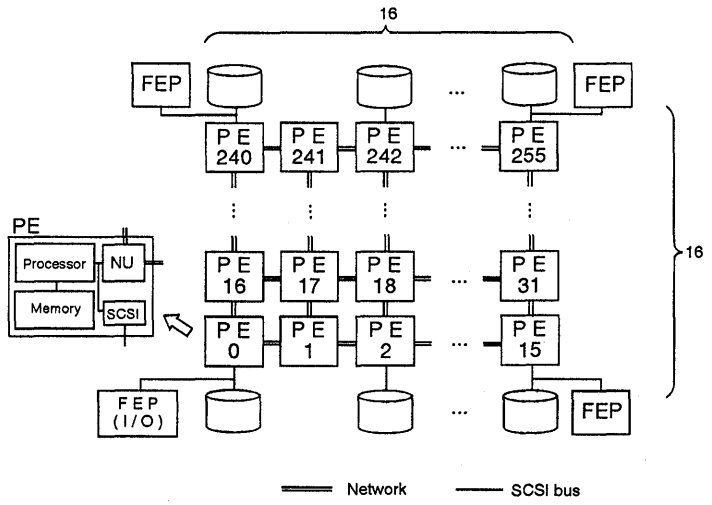


Figure 4: Overview of PIM/m Architecture

PIM/c : PIM/c also takes the multi-cluster architecture including 256 processors in total. A cluster contains eight processors. 32 clusters are connected with a crossbar switch network [Nakagawa *et al.* 1992].

PIM/k : PIM/k focuses on architectural research within a cluster. Hierarchical cache system has been investigated to connect larger number of processors in a cluster [Sakai *et al.* 1991]. Four processors share a local bus and second cache. They form a mini-cluster. Four mini-clusters are connected to a shared memory-bus and shared memory (Figure 5).

PIM/i : PIM/i is also a research use system. LIW-type instruction set and cache protocol with broadcasting type has been investigated [Sato *et al.* 1992].

The global configuration of five PIMs are summarized in table 1.

Specifications of components, that are processors, networks, and cache systems, will be reported in the following subsections.

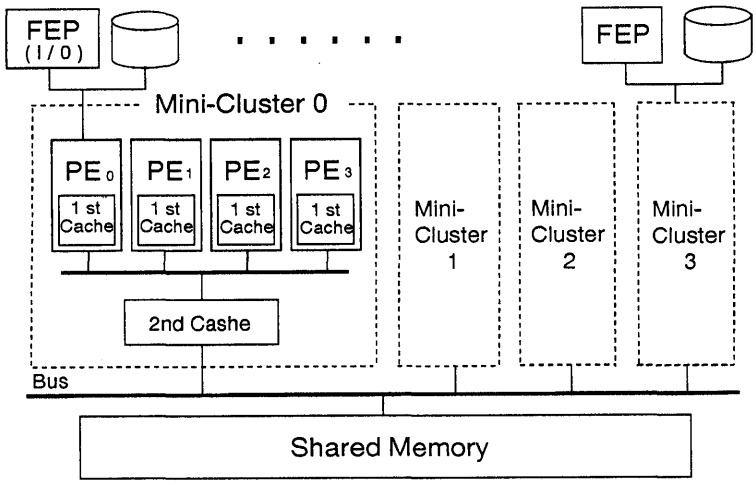


Figure 5: Overview of PIM/k Architecture

Table 1: Global Configuration

	Topology	Number of Clusters	Total Number of PEs	Memory Size/Cluster
PIM/p	hypercube × 2	64	512	256 MB
PIM/m	mesh	256	256	80 MB
PIM/c	crossbar	32	256	160 MB
PIM/k	—	1 †	16	1 GB
PIM/i	—	2	16	320 MB
Multi-PSI/V2	mesh	64	64	80 MB

(† : four mini-clusters included)

4.2 Processing Element

Since KL1 implementation requires frequent runtime type checking, all CPUs of PIM models are designed as the tagged-architecture similar to the Multi-PSI.

PIM/p, PIM/i and PIM/k have RISC-like instruction set whereas PIM/m and PIM/c have CISC-like micro programmable instruction set (Table 2). The former processors execute machine instructions which are at a level still lower than KL1-B. The latter processors interpret KL1-B code by horizontal micro program.

The CPU of PIM/p [Kumon *et al.* 1992] has a unique feature called *macro-call* [Shinogi *et al.* 1988] instructions for light-weight subroutine calls. The instructions enable the size of compiled user program codes to be kept small and to reduce the overheads of subroutine calls. It also has some more instructions dedicated to KL1 implementation, such as dereference instructions and MRB [Chikayama and Kimura 1987] incremental garbage collection instructions. The CPU takes four-stage pipeline structure.

The CPU of PIM/m [Nakashima *et al.* 1992] is a microprogram controlled processor with five-stage pipelining. The instruction set is KL1-B itself, which is binary compatible with Multi-PSI/V2. Sophisticated data type checking and the automatic dereference mechanism are special features.

The CPU of PIM/i tries the LIW(long instruction word)-type instruction set.

4.3 Network

Networks are summarized in table 3.

In PIM/p, each processor has a NI and four NIs are connected to a router. The router works as a node in the network. There are two hypercube networks to attain large band width.

PIM/m has a two dimensional mesh network, similar to the Multi-PSI. The networks of PIM/p and PIM/m realize so-called the worm-hole routing.

Table 2: Specification of Processing Element

	Instruction set	Cycle time	LSI fabrication	Line interval
PIM/p	RISC + macro instruction	60 nsec †	standard-cell	0.96 μm
PIM/m	CISC (micro programmable)	65 nsec	standard-cell	0.8 μm
PIM/c	CISC (micro programmable)	50 nsec †	gate-arrays	0.8 μm
PIM/k	RISC	100 nsec	custom	1.2 μm
PIM/i	RISC	100 nsec †	standard-cell	1.2 μm
Multi-PSI/V2	CISC (micro programmable)	200 nsec	gate-arrays	2.0 μm

(† are design specifications. They are under testing with longer cycle time.)

Table 3: Network

	# PEs in a cluster	# NIs in a cluster	Transfer Rate †
PIM/p	8	8	33 MB/sec ‡ ×2
PIM/m	1	1	8 MB/sec
PIM/c	8	1	40 MB/sec ‡
PIM/k	16	—	—
PIM/i	8	1	—
Multi-PSI/V2	1	1	10 MB/sec

(PE = processing element, NI = network interface)
(†: per channel, full duplex ‡: design specifications)

PIM/c has one special processor named cluster controller in each cluster. The cluster controller is connected to a shared bus and works as a network interface to a crossbar network. The cluster controller has overall responsibility for network communications.

4.4 Cache System

Since KL1 programs arise asynchronous communications among processors very frequently, shared bus traffic tends to become very heavy. To solve this problem, an optimized coherent cache protocols were designed [Goto *et al.* 1989][Matsumoto *et al.* 1987], which can keep the locality high and reduce the shared bus traffic [Nishida *et al.* 1990]. All PIMs have write-back type coherent cache protocols (Table 4). Low cost locking mechanisms are also supported with utilizing the cache block status.

5 KL1 Language Implementation

KL1 language has many beneficial features to write efficient concurrent and parallel programs of *the dynamic and non-uniform problems*, which was explained in section 3.4. The KL1 implementation is focused to realize the execution efficiency of those language features. This section looks at the language implementation methods and techniques briefly, that correspond to the implementation features presented in section 3.5. The purpose of this section is to give a concrete image of several key features of the KL1 implementation. Detailed information are presented in [Hirata *et al.* 1992] [Nakajima 1992].

5.1 Execution Model of KL1

For the help of getting the image, the execution model of KL1 is shown briefly. KL1 program is made up of a collection of clauses, whose form is:

$$\underbrace{H :- G_1, ..., G_m}_{guard\ part} \mid \underbrace{B_1, ..., B_n.}_{body\ part}$$

where H is the *head*, G_i the *guard goal*, that are collectively called the *guard part*. The B_i are the *body goals* and the vertical bar (|) is the *commitment operator*.

The guard part can be considered as a pattern match and condition tests. If there are alternative clauses, their guard parts are tested sequentially. When a clause succeeds the pattern match and the condition tests, the clause commits. The caller goal is reduced to the body

Table 4: Specification of Cache System

	Coherence Control		Mapping	Cache Size	
	Protocol	# States †		Instruction	Data
PIM/p	invalidation	4	4 way	64 KB	
PIM/m	—	—	direct-map	5 KB	20 KB
PIM/c	invalidation	5	2 way	80 KB	
PIM/k	hierarchical invalidation	4	(1st) direct-map	128 KB	256 KB
			(2nd) 4 way	1 MB	4 MB
PIM/i	broadcasting	6	direct-map	160 KB	160 KB
Multi-PSI/V2	—	—	direct-map	20 KB	

(† does not include *locking* state.)

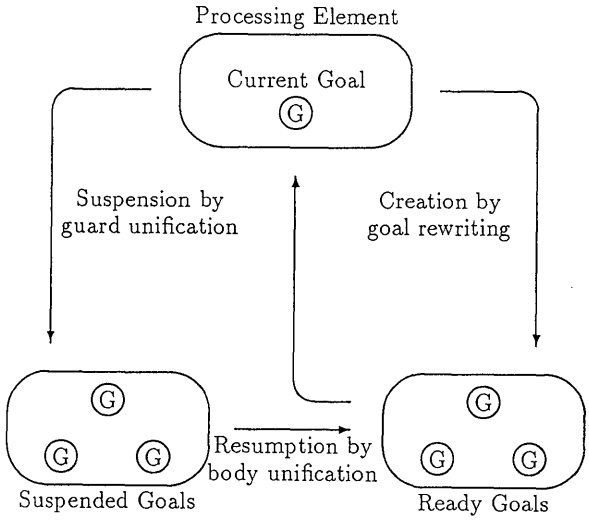


Figure 6: Execution Model of KL1

goals of the committed clause. These body goals are executed concurrently (AND-parallel). A KL1 clause can be considered as a rewrite rule, which rewrites the caller goal to the body goals.

An execution model of KL1 is shown in Figure 6. There is a goal pool which holds the ready goals to be rewritten. One of ready goals is taken from the goal pool for the execution, which is the current goal. When there is a clause, which matches the current goal and succeeds the condition tests, the current goal is rewritten. The rewritten goals are placed back to the goal pool.

Goals may have common variables, that are used for the communication and synchronization. Let us assume that there are two goals sharing a logical variable. A body unification, produced in a goal rewriting, can instantiate the variable. Guard unifications, that appear in a execution of the other goal, test the instantiated value of the variable. This is the communication between the goals. When the variable is not instantiated before the guard unification, and no other clause can commit, the current goal is suspended. Instantiation of the variable resumes the suspended goal. This is the synchronization [Ueda and Chikayama 1990].

5.2 Supports for the Implicit Communication

There are several important mechanisms that realize the implicit communication between computation nodes.

Let us assume that there are two goals sharing a variable in a computation node. Each goal has a reference to the variable. When a goal is sent to the other computation node, a remote reference has to be generated implicitly. The implicit communication between the goals in the different nodes will be performed along with this remote reference.

The important mechanisms are shown briefly.

5.2.1 Global Name Space

The implicit reference management across the computation nodes are supported for logical variables, structured data and program code. It is a support of the virtual global name space on a distributed memory hardware.

The **export/import tables** realize the feature. The export/import tables are the indirect reference tables that separate the local address space in a computation node and the global space for the remote references (Figure 7). The remote reference (external reference) is identified by the pair ⟨A,e⟩, where A is the node number in which the referenced data resides, and e is the entry number of the export table. Registration to the tables are performed dynamically when a new remote reference is made [Ichiyoshi *et al.* 1987].

The entry number e does not change even when a local garbage collection occurs which moves the location of the exported cell. When a **duplicated exportation/importation** occurs, the same table entry number is used (reducing a new registration to the table)

which eliminates useless data transfer between nodes [Ichiyoshi *et al.* 1988].

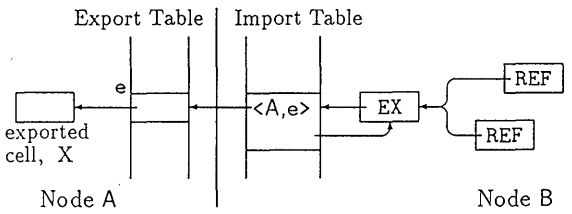


Figure 7: Export and Import Tables

5.2.2 Implicit Data Transfer

Data Transfer by Unifications: The implicit data transfer between computation nodes is initiated by unifications.

A guard unification tries to test an instantiation of a logical variable. When it is an external reference (EX in Figure 7), a read request message, `%read(X, ReturnAddress)`, is sent to the node A. Where `X` is the external reference ⟨A,e⟩, and `ReturnAddress` is a newly created export table entry in the node B.

The goal execution, which initiated the guard unification, is suspended when no other clause can commit.

When the referenced cell has a concrete value `V`, it is returned by the message, `%answer_value( ReturnAddress, V )`. The message resumes the suspended goal, which waits for the value `V`. If the referenced cell is not bound to a fixed value, the read request is suspended until the variable is instantiated.

When a body unification tries to unify a remote cell X with a term Y, a message `%unify(X, Y)` is sent to the referenced cluster. When Y is an atomic data or a structure, a simple data transfer occurs.

The unifications between two uninstantiated variables in different clusters may make reference loops between clusters. This problem can be solved by controlling the direction of reference pointers [Ichiyoshi *et al.* 1988].

Lazy Transfer: When a structured data is transferred between nodes, one-level transfer is performed. The components of a structure may be atomic data or nested structures. The atomic data are copied and transferred directly, while the nested structures are remained as pointers and transferred as external references. This is called the one-level transfer. The policy is that the data transfer should be delayed as lazily as possible, until the data is really needed for some operation.

Code Transfer: Program codes are handled as large structured data. They are loaded on one cluster by a loader program at first. Any KL1 goal hold the reference to the corresponding code object. When a goal is sent to a cluster and the cluster does not contain the corresponding code object, the goal execution is suspended and the code is dynamically transferred from the cluster which is pointed by the external reference held in the goal.

5.3 Small-Grain Concurrent Processes

5.3.1 Process Group Management

KL1 goals can be considered as lightweight processes. For the efficient parallel processing, a user task have to include a lot of lightweight processes. It is needed for the parallel operating system that a group of goals (lightweight processes) can be handled all together as a task. The **shoen** supports the meta control facilities of execution control, resource management and status monitoring for the goal group.

Shoen and Foster Parent: Any goals have to belong to a certain *shoen*. The *foster-parent* `fp` is a proxy *shoen*, which is created in every computation nodes where the goals of the *shoen* are executed. Each goal points their foster-parent in the node, and test the request for meta-controls in a certain interval (e.g. in every goal reductions). Figure 8 shows the relationship among *shoens*, foster-parents and goals.

A *shoen* and a foster-parent keep their environments, such as status, resources, and the number of goals. Foster-parents reduce the communication between each goal and their *shoen*, to avoid an access bottleneck at the *shoen*.

Termination Detection: The termination detection of a goal group is one of the difficult subjects in parallel computation systems, especially when messages may be in transit on the network. Even if all the foster parents report their terminations, the *shoen* should not terminate when there are goals in transit.

One of the solutions is the *Weighted Throw Counting* (WTC) scheme [Rokusawa *et al.* 1988], which is an application of the *Weighted Reference Counting* (WRC) scheme [Watson and Watson 1987].

5.3.2 Goal Scheduling

The goal scheduling, discussed here, is a different concept with the goal group management by *shoen*. The goal scheduling is the state transition management of each goals, among *ready*, *execution*, and *suspension* states. Execution priority is also managed.

Basic Goal Scheduling Scheme: The *ready* goals in a computation node are linked into a list forming a *ready-goal-stack*. In principle, a current goal is popped from the

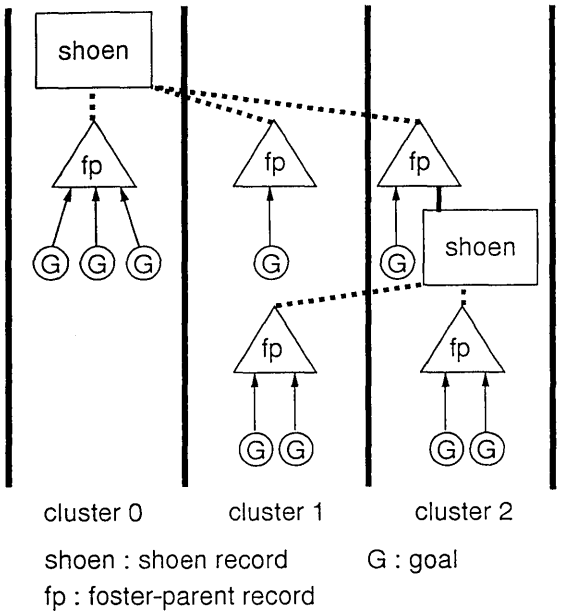


Figure 8: Relationship of *shoen* and foster-parents

ready-goal-stack, then the goal rewriting is performed. The rewritten goals are pushed to the ready-goal-stack, which is the **depth-first scheduling** in a computation node.

When any unification suspends, the goal is linked as a suspended goal to the variable which caused the suspension. Here, the **non-busy waiting** method has been adopted. That is, the suspended goal is not scheduled until the variable will be instantiated. When a suspended goal is resumed, it is linked to the ready-goal-stack again.

Execution priority of goals can be specified by pragmas. The ready-goal-stack is managed with the priority of goals.

Goal Distribution within a Cluster: An automatic load balancing scheme is tried within a cluster. An individual ready-goal-stack is provided for the highest priority goals in each processing element, to avoid conflicts of access to the common goal-stack [Sato *et al.* 1987]. The highest-priority goals are distributed to keep the processor loads in good balance [Hirata *et al.* 1992].

Inter-cluster Goal Distribution: A body goal, goal@node(CL), is thrown with a message %throw to a node CL when the clause commits. The node (more precisely, a certain processing element in the cluster CL), that received the %throw message, links the goal to its ready-goal-stack as well as to the foster-parent. If there is no foster-parent, one will be created on the spot.

5.4 Memory Management

Memory management like dynamic memory allocation, reclamation, and garbage collection are indispensable for concurrent symbolic processing languages.

5.4.1 Incremental Garbage Collection by MRB

The MRB method is a subset of the reference counting scheme which maintains one-bit information in pointers indicating whether the pointed data object has multiple references to it or not [Chikayama and Kimura 1987] [Inamura *et al.* 1988]. Garbage cells that have only a single reference can be reclaimed incrementally.

The MRB is also useful to optimize the updating of structured data. Structured data must be copied in principle when it is updated partially, because of the single-assignment feature. However, it can be rewritten destructively when the structure has only a single reference, keeping a semantics of the single-assignment language.

5.4.2 Garbage Collection within a Cluster

Another garbage collection is implemented, which is performed locally within a cluster accompanied with the incremental garbage collection by MRB. Because the MRB scheme leaves some garbages.

So-called *stop and copy* scheme is adopted basically. The parallel mechanism has been investigated to collect garbages by all processing elements in parallel in a cluster [Imai and Tick 1991].

5.4.3 Inter-Cluster Garbage Collection by WEC

An incremental inter-cluster garbage collection scheme, the weighted export counting (WEC) scheme is employed [Ichiyoshi *et al.* 1988]. It is an application of the weighted reference counting (WRC) scheme [Watson and Watson 1987]. The scheme has several advantages. One is the incremental garbage collection capability with fewer message exchanges compared with the full reference counting. The other is also a capability of reducing the messages for the case when a imported data has to be exported again to the different clusters.

5.5 Abstract Instruction Set KL1-B

KL1-B is the abstract instruction set which is common in PIM models. The role of KL1-B is similar to that of WAM [Warren 1983]. An explanation of each KL1-B instruction can be found in [Kimura and Chikayama 1987].

Most of the KL1 implementation schemes, presented in previous sections, are realized as runtime routines that are invoked by certain KL1-B instructions implicitly.

The KL1 compiler for PIM has two phases. The first phase compiles a KL1 program into an KL1-B code. The second phase translates the KL1-B code into a native code, making a linkage with runtime routines.

6 Measurements and Evaluation

This section describes some measurements results and evaluations for the parallel inference machines and the language system. The measurements focused on a low-cost implementation of small-grain concurrent processes and remote synchronization and communication. Measurements on a few benchmark programs are also reported, including the most recent measurements on PIM/m.

6.1 Measurements and Evaluation on the Multi-PSI/V2

The KL1 language implementation includes so-called OS kernel functions, as shown in section 3.5. Most of the implementation features, that were presented in section 5, concern with the OS kernel functions. Efficient implementations of these functions enable the actual use of the beneficial features of KL1 language (presented in section 3.4) to write efficient parallel programs of *the dynamic and non-uniform problems* for large-scale parallel machines.

The actual execution cost of some of these functions have been measured on the Multi-PSI/V2. Goal scheduling cost within a computation node, communication cost between nodes, and communication overhead in benchmark programs are reported. Measurements results shows the quite low-cost implementations.

Note that the Multi-PSI/V2 has a mesh structure with 64 processing elements (PEs). There are 64 computation nodes each of which is one PE.

6.1.1 Goal Scheduling Cost in a Node

Goal scheduling and synchronization cost within a processing element (PE) have been measured [Onishi *et al.* 1990].

The **enqueue and dequeue** cost of a simplest goal is 5.4 μs (27 micro-instruction steps). When a goal is rewritten to several goals in a goal reduction, they are pushed on the ready-goal-stack once (except for one goal which can be executed directly). The enqueue and dequeue cost is the summation of the pushing and popping cost of a goal to the ready-goal-stack. The enqueue and dequeue cost can be considered as a part of the *process fork cost.*

The **single-suspension** cost of a simple goal is 14 μs (70 steps). When a goal is suspended waiting for a variable instantiation, the goal is hooked to the variable cell. When the variable is instantiated, the goal becomes executable and is pushed on the ready-goal-stack. The single-suspension cost is a summation of the hook, enqueue, and dequeue cost. The single-suspension cost can be considered as the *synchronization cost* between processes in a processor.

The **two-way multiple-suspension** cost of a simple goal is 28 μs (140 steps). A goal can wait for the variable instantiation of several different variables. The first instantiation resumes the goal execution. If the instantiation causes a comitment of a clause, the other waiting conditions are thrown away. The two-way multiple-suspension is a case of two variables. The feature is a combination of the indeterminacy and the synchronization. Cost increase from the single-suspension corresponds to the implementation cost of the *indeterminacy.*

These low-cost implementations encourage the actual use of a lot of small-grain processes. These costs of the goal scheduling also give a guideline for the lower bound of process grain size for efficient execution within a computation node.

6.1.2 Communication Cost Between Nodes

Cost of the communication primitives have been measured on the Multi-PSI/V2 system [Nakajima and Ichiyoshi 1990]. A goal sending to another PE (a remote call of a lightweight process) is realized by `%throw_goal` message. Inter-PE reading of values (used for remote synchronization and communication) is realized by `%read` & `%answer_value` protocols.

Figure 9 shows the cost of handling those three messages at both sending and receiving PE.

The
cost is broken down into three parts. `Encode/decode KL1 term, etc.` is for encoding and decoding message packets to/from internal representations of KL1 term. It also includes the maintenance of the export/import tables and the foster parent records (c.f. section 5). It is the essential part of the message handling.

`Basic message handling routine` in Figure 9 corresponds to the simple data conversion between 40-bit tagged words and byte-serial messages. The routine includes data transfer to/from the hardware buffer. The cost can be potentially reduced by hardware supports. `Copy_RPKB` stands for copying a message packet from the hardware buffer to the software buffer. It is only executed when the hardware buffer tends to be full.

The network transfer speed is 0.2 μs/byte. It takes below 1 μs to hop one network node. It means that the message handling cost, just explained before, is dominant in the communication cost.

`Send_throw` (a) shows the cost of sending a 65 byte `%throw_goal` message containing a goal with three arguments. It takes 419 micro-instruction steps or 85 μs (cycle time = 200 ns). `Receive_throw` (b) shows the cost of receiving the same `%throw_goal` message and storing it in a goal stack.

The bar graphs (c), (d), (e) and (f) describe the cost of sending and receiving a `%read` message and

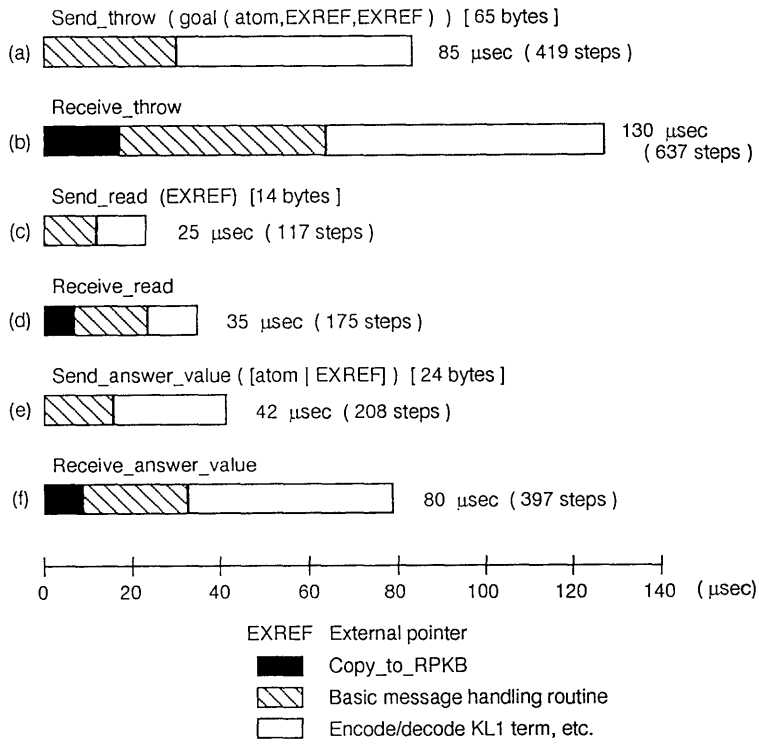


Figure 9: Message Handling Cost

Table 5: Message Frequency and Reductions

Pentomino (39.3 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (sec)	54.63	14.62	4.35
total reductions (×1000)	8,317.	8,332.	8,340.
reductions/sec (KRPS)	152.2	570.1	1,919.4
reductions/msg	221.	108.	88.
msg bytes/sec (×1000)	14.5	108.1	440.5

Bestpath (23.4 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (sec)	10.655	4.062	1.691
total reductions (×1000)	987.7	1213.6	1,505.2
reductions/sec (KRPS)	92.7	298.8	890.1
reductions/msg	21.9	11.7	6.2
msg bytes/sec (×1000)	114.0	692.5	3,854.3

(KRPS: Kilo Reductions Per Second)

Table 6: Single Processor Performance of PIM/m

benchmark	condition	PIM/m	Multi-PSI/v2	$\frac{Multi-PSI/v2}{PIM/m}$
append	1,000 elements	1.63 msec	7.80 msec	4.8
best-path	90,000 nodes	142 sec	213 sec	1.5
pentomino	8 × 5 box	107 sec	240 sec	2.2
15-puzzle	5,885 K nodes	9,283 sec	21,660 sec	2.3

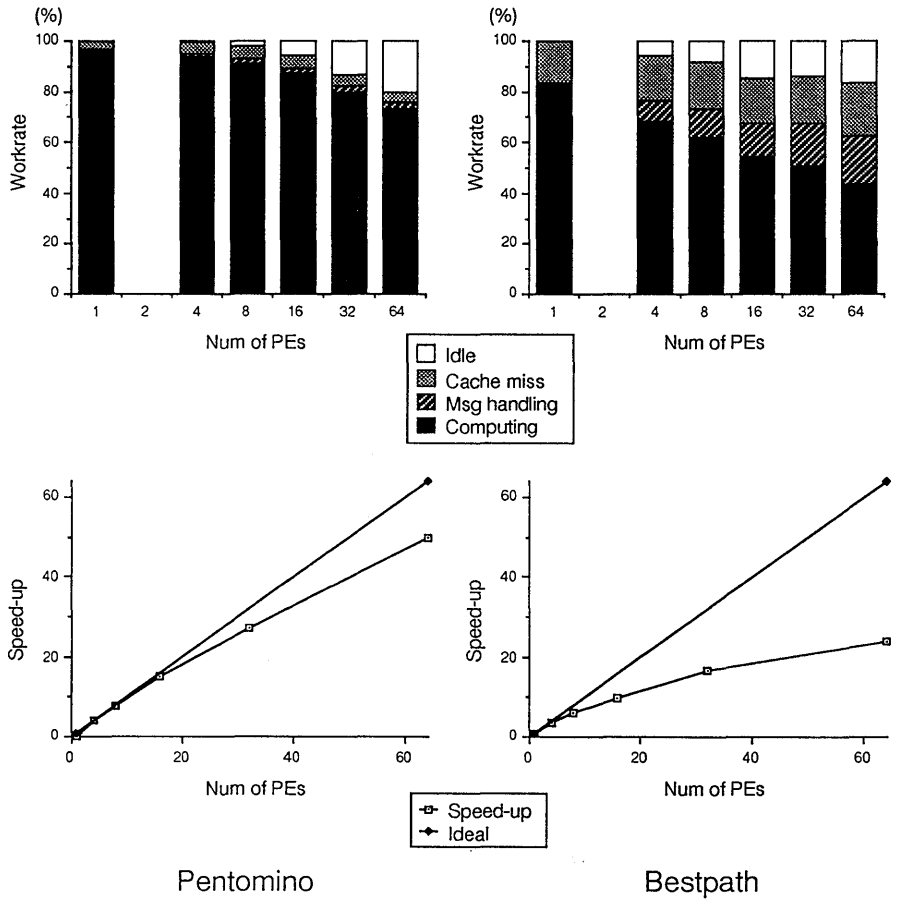


Figure 10: Decomposition of Processor Time and Speed-up

Table 7: System Performance on Pentomino (8 × 5 box)

No. of PEs	PIM/m		Multi-PSI/v2		$\frac{Multi-PSI/v2}{PIM/m}$
	Time	Speedup	Time	Speedup	
256 PE	1,124 ms	95.41			
128 PE	1,290 ms	83.13			
64 PE	2,162 ms	49.60	4,679 ms	51.20	2.16
32 PE	3,694 ms	29.03	8,278 ms	28.94	2.24
16 PE	6,910 ms	15.52	15,686 ms	15.27	2.27
1 PE	107,238 ms	1.00	239,545 ms	1.00	2.23

`%answer_value` message.

Sending and receiving cost of the `%throw_goal` message, 215 μs (1056 steps) in total, can be considered as the cost of a process fork to a different PE, or a remote procedure call. Cost of the `%read` and `%answer_value` messages, 182 μs (897 steps) in total, correspond to the cost of the remote synchronization.

Comparing these value with the cost of local operations in the previous section, the remote synchronization takes around 10 times higher cost than local. The remote procedure call costs more but below 40 times of the local process fork. These *remote/local ratio* seems low enough

Table 8: System Performance on Pentomino (10 × 6 box)

No. of PEs	PIM/m		Multi-PSI/v2		$\frac{Multi-PSI/v2}{PIM/m}$
	Time	Speedup	Time	Speedup	
256 PE	103,655 ms	234.29			
128 PE	188,452 ms	128.87			
64 PE	359.268 ms	67.60	886,325 ms		2.47
32 PE	694,553 ms	34.96	1,729,430 ms		2.49
16 PE	1,367,240 ms	17.76			
1 PE	24,285,015 ms	1.00			

to encourage the small-grain concurrent processing between PEs. Measurements of the communication cost give a guideline for the process grain size (communication rate) to keep the communication overhead low. When a process garin size decreases, becoming close to the communication cost, communication overhead increases significantly (close to 50% of CPU time).

6.1.3 Measurements on Benchmark Programs

Benchmark Programs: The followings are the two benchmark programs used here.

- **Pentomino:** A program to find out all solutions of a packing piece puzzle (Pentomino) by exploring the whole OR tree. Two-level dynamic load balancing is employed [Furuichi *et al.* 1990].
- **Bestpath:** A 160 × 160 grid graph is given together with non-negative edge costs. The program determines the lowest cost path from a given vertex to all vertices of the graph by performing a distributed shortest path algorithm [Wada and Ichiyoshi 1990]. The vertices are represented by KL1 processes, and they exchange shortest path information along the edges. 25,600 small processes work cooperatively.

Message & Reduction Profile: Table 5 shows the execution time, the reduction and message rates, etc. [Nakajima and Ichiyoshi 1990]. Average time of one reduction in a PE is an inverse of the KRPS value. 25 μs (127 steps) in Pentomino, and 43 μs (214 steps) in Bestpath. They are almost the grain size of concurrent processes in a PE. The message sending rates on 64 PEs are: one message per 88 reductions in Pentomino, and one per 6 reductions in Bestpath.

The average network traffic was repoted in [Nakajima and Ichiyoshi 1990], calculated from these figures. Relative to the 10 Mbyte/s network channel bandwidth, the average traffic on a channel is very small: 0.08% (Pentomino) and 0.3% (Bestpath) of the bandwidth.

Communication Overhead: Profiling data of processor execution has been measured on the two benchmark programs [Nakajima 1992]. The execution time is broken down into the four categories in Figure 10: computing time (reduction operations), message handling time, cache-miss penalty, and idling time. The average of all PEs are shown in the bar graph. The resultant speed-up is also shown with the ideal one.

Two-level dynamic load distribution is used in Pentomino. Several thousands small processes are distributed to 64 PEs in 4.35 seconds adaptively. The graph shows low communication overhead and good speedup. The degradation of processor workrate in 64-PE execution is mainly caused by the latency of load feeding to PEs.

In Bestpath, 25,600 small processes are distributed statically on 64 PEs. They exchange messages to perform an distributed algorithm. The inter-PE communication and the cache-miss penalty degrade the performance because of the high communication rate and the large working set. As the number of PEs grows, the grid graph is divided into smaller blocks to keep the workrate high, and it makes the inter-PE communication rate higher. Best path includs speculative computation, which increases with the large number of PEs. It causes lower speedup than a calculated value from the processor workrate.

Measurements results in table 5 and Figure 10 show the actual communication rate and communication overhead. Programmers can use relatively large communication rate, one message per 6 reductions (measured in Bestpath), with non-large CPU overhead of approximately 15%. Considering a network load of 0.3% at that time, it is observed that CPU load (15% at that time) will limit the communication band width when communication rate increases. The language implementation, which supports the global name space on a distributed memory hardware, tends to increase the CPU load concerned with network communication.

6.2 Preliminary Measurements on the PIM

6.2.1 Single Processor Performance

Table 6.1 shows the single processor performance of PIM/m for four benchmarks. The table also includes the performance of Multi-PSI/V2 and the ratio of PIM/m and Multi-PSI/V2 (M/P-speedup).

M/P-speedup is 1.5 to 2.3 in average. Programs with large working set tends to show low M/P-speedup.

6.2.2 System Performance

Table 7,8 show the preliminary measurements of system performance on PIM/m. The benchmark program is Pentomino.

Speedup saturation in Table 7 is caused by small problem size. Better speedup (234 folds speedup with 256 processors) was attained with larger problem in Table 8. It is also surprising that the small problem (executed in 1.1 second) show 95 folds speedup, which uses the multi-level dynamic load distribution distributing several thousands of small processes. The facts shows an efficient language implementation suitable to handle a lot of small-grain processes with less overhead.

7 Conclusion

This paper described two subjects. One is an overview of the research and development on the parallel inference machine PIM and the language implementation of the kernel language KL1, a concurrent logic programming language.

The other is the clarification of the features and advantages of KL1 language, its parallel implementation, and the hardware architecture from the viewpoint that the features are suitable and may be indispensable for efficient parallel processing of *the dynamic and non-uniform problems* with large computation. Knowledge processing is included in the problem domain. These problems have not been covered by commercial parallel machines and their software systems that target the scientific computation. The PIM system focuses on this new domain of parallel processing.

PIM is a distributed memory MIMD machine with a global view, connecting a maximum of 512 processors. It includes shared-memory substructures. Many component technologies have been developed that support efficient parallel processing on the target problem domain, especially on symbolic processing.

KL1 language also has very strong features for efficient programming and execution of the dynamic and non-uniform large problems. Major features are (1) small-grain concurrent processes, (2) implicit synchronization and communication, (3) separation of concurrency design and mapping (load allocation and scheduling), etc. They support highly concurrent programming with complex structures and support large flexibility for load balancing. The efficient language implementation made actual use of the language features possible. The PIM and KL1 system have realized a strong research and development environment for parallel software in that problem domain.

Measurements and evaluations showed a very low-cost language implementation for handling small-grain concurrent processes and their remote communications. Good speedup by parallel processing on benchmark programs was also reported. A lot of small-grain processes were handled during this processing. These results prove the efficiency and usefulness of the system to *the dynamic and non-uniform problems.*

Further measurement and evaluation is continuing, and the results of this will be reported soon. On the other hand, many problems of parallel software remain unsolved. Continuous research must be carried out to construct the real technology of large-scale parallel processing for *the dynamic and non-uniform problems* including the knowledge information processing in the 21st century. The parallel inference machine PIM and the KL1 language system will be utilized as the best research environment.

Acknowledgment

The R & D of PIM system have been carried out by researchers in the first research laboratory and cooperating companies, supported with valuable suggestions and helps by members of the second, seventh and the other ICOT laboratories and the PIM working group. The author would like to thank all of these people for their continuous efforts and cooperation.

References

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. of the Fourth Int. Conf. on Logic Programming*, 1987, pp.276–293.

[Chikayama 1992] T. Chikayama. Operating System PIMOS and Kernel Language KL1. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the Multi-PSI. In *Proc. of PPoPP'90*, pp. 50–59, 1990.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the*

Int. Conf. on Fifth Generation Computer Systems, ICOT, Tokyo, 1988, pp.208–229.

[Goto *et al.* 1989] A. Goto, A. Matsumoto and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *Proceedings of 16th Annual International Symposium on Computer Architecture*, pages 25 – 33, Jerusalem, Israel, 1989.

[Hirata *et al.* 1992] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase and K. Rokusawa. Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Ichiyoshi *et al.* 1987] N. Ichiyoshi, T. Miyazaki and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proceedings of Fourth International Conference on Logic Programming*, pages 257–275, University of Melbourne, MIT Press, 1987.

[Ichiyoshi *et al.* 1988] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *New Generation Computing*, Ohmsha Ltd. 1990, pp.159–177.

[Ichiyoshi 1989] N. Ichiyoshi. Parallel logic programming on the Multi-PSI. ICOT Technical Report TR-487, ICOT, 1989. (Presented at the Italian-Swedish-Japanese Workshop '90).

[Imai *et at.* 1991] A. Imai, K. Hirata and K. Taki. PIM Architecture and Implementations. In *Proc. of Fourth Franco Japansese Symposium*, ICOT, Rennes, France, 1991.

[Imai and Tick 1991] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *ICOT Technical Report*, TR-650, 1991. (To appear in IEEE Transactions on Parallel and Distributed Systems)

[Inamura *et al.* 1988] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Techiniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. of the North American Conf. on Logic Programming*, 1989, pp. 907–921 (also *ICOT Technical Report*, TR-466, 1989).

[Kimura and Chikayama 1987] Y. Kimura and T. Chika-yama. An Abstract KL1 Machine and its Instruction Set. In *Proc. of Symposium on Logic Programming*, 1987, pp.468–477.

[Kumon *et al.* 1992] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa and K. Hirano. Architecture and Implementation of PIM/p. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Masuda *et al.* 1988] Y. Masuda, Y. Ishizuka, Y. Iwayama, K. taki and E. Sugino. Preliminary Evaluation of the Connection Network for the Multi-PSI System. In *Proc. Europian Conference on Artificial Intelligence 1988 (ECAI-88)*, August 1988.

[Matsumoto *et al.* 1987] A. Matsumoto, T. Nakagawa, M. Sato, K. Nishida and A. Goto. Locally Parallel Cache Design Based on KL1 Memory Access Characteristics. ICOT Technical Report 327, 1987.

[Nakagawa *et al.* 1989] T. Nakagawa, A. Goto and T. Chikayama. Slit-Check Feature to Speed Up Interprocessor Software Interruption Handling. In *IPSJ SIG Reports*, 89-ARC-77-3, 1989 (*In Japanese*).

[Nakagawa *et al.* 1992] T. Nakagawa, N. Ido, T. Tarui, M. Asaie and M. Sugie. Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. of the Sixth Int. Conf. on Logic Programming*, 1989, pages 436–451.

[Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI. In *ICOT TR-531*, 1990.

[Nakajima 1992] K. Nakajima. Distributed Implementation of KL1 on the Multi-PSI. In *Implementation of Distributed Prolog*, edited by P. Kacsuk and M. Wise, John Wiley & Sons, Ltd., 1992.

[Nakashima and Nakajima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987, pp 104–113.

[Nakashima *et al.* 1992] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y. Inamura, S. Onishi and K. Masuda. Architecture and Implementation of PIM/m. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Nishida *et al.* 1990] K. Nishida, Y. Kimura, A. Matsumoto and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *Proc. of the Seventh Int. Conf. on Logic Programming*, 1990, pages 83–95.

[Nitta *et al.* 1992] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Onishi *et al.* 1990] S. Onishi, Y. Matsumoto, K. Nakajima and K.Taki. Evaluation of the KL1 Language System on the Multi-PSI. In *Proc. of Workshop on Parallel Implementation of Languages for Symbolic Computation*, July 30–31, 1990, Oregon, USA. Also ICOT TR-585.

[Rokusawa *et al.* 1988] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proc. of the 1988 Int. Conf. on Parallel Processing*, Vol. 1 Architecture, 1988,pp.18–22.

[Rokusawa and Ichiyoshi 1992] K. Rokusawa and N. Ichiyoshi. A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting. In *Proc. of Sixth Int. Parallel Processing Symposium*, IEEE, 1992.

[Sato *et al.* 1987] M. Sato, A. Goto, et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987.

[Sato and Goto 1988] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proc. of IFIP Working Conf. on Parallel Processing*, 1988, pp. 305–318.

[Sato *et al.* 1992] M. Sato, K. Takeda and T. Ohara. Design of the Parallel Inference Machine PIM/i Processor. In *Trans. of IPSJ*, Vol.33, No.3, 1992, pp. 278–287 (*In Japanese*).

[Shinogi *et al.* 1988] T. Shinogi, K. Kumon, A. Hattori, A. Goto, Y. Kimura and T. Chikayama. Macro-call Instruction for the Efficient KL1 Implementation on PIM. In *Proceedings of the International Conference on Fifth Generation Computing Systems 1988*, Tokyo, Japan, pages 953–961, 1988.

[Takagi and Nakase 1991] T. Takagi and A. Nakase, Evaluation of VPIM: A Distributed KL1 Implementation – Focusing on Inter-cluster Operations –, In *IPSJ SIG Reports*, 91-ARC-89-27, 1991 (*In Japanese*).

[Takeda *et al.* 1988] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Taki *et al.* 1984] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1984*, pp.398–409, Tokyo, Nov. 1984.

[Taki 1988] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI System. In *Programming of Future Generation Computers*, K.Fuchi and M.Nivat (Editors), pages 411–426, Elsevier Science Publishers B.V., North Holland, 1988.

[Uchida *et al.* 1988] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of The FGCS Project. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988*, pp.16–36, Tokyo, Nov. 1988.

[Uchida 1992] S. Uchida. Summary of the Parallel Inference Machine and its Basic Software. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Ueda 1986] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Technical Report 208, 1986.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, (33)6, 1990, pp.494–500.

[Wada and Ichiyoshi 1990] K. Wada and N. Ichiyoshi. A study of mapping locally message exchanging algorithms on a loosely-coupled multiprocessor. ICOT Technical Report TR-587, 1990.

[Warren 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

[Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proc. of Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, 1987, pp.432–443.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Operating System PIMOS and Kernel Language KL1

Takashi Chikayama
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan
chikayama@icot.or.jp

Abstract

The Fifth Generation Computer Systems (FGCS) project is a national project of Japan, aiming at establishing the basic technology required for high performance knowledge information processing systems. The parallel inference system subproject is aiming at establishing parallel processing hardware technology for massive processing power and software technology for effective utilization of such hardware in the knowledge information processing field. The basic software system is responsible for providing a programming language suited for describing knowledge information processing applications software and providing a comfortable environment for program execution and software development on highly parallel computer systems.

A concurrent logic language with extensions to control program execution on parallel hardware was designed as the *kernel* language of the system. An operating system that provides a comfortable environment for parallel application software development was designed and implemented in the kernel language. This paper gives an overview of the research and development in this area in the FGCS project.

1 Introduction

The fifth generation computer systems project is a national project of Japan, aiming at establishing the basic technology required for high-performance knowledge information processing systems. The most important technologies to be provided to attain the final objective of the project are the following two.

- Problem solving methods for knowledge information processing
- Processing power for implementation of the above methods

The parallel inference system subproject is aiming at establishing both hardware and software technologies for the latter.

With the recent evolution of the hardware technology, multiprocessor systems are expected to be advantageous not only in absolute processing power but also in cost effectiveness early in the next century. There seems to be no other technology than multiprocessing to provide the computational power required for high-performance knowledge information processing systems.

The software technology for parallel processing, on the other hand, is still quite premature. In particular, the technology for building parallel software to solve complicated problems in the area of knowledge processing is far from satisfactory yet. This, we think, is at least partly due to the problems in the approach to the parallel software technology conventionally taken, that is, trying to augment already available sequential processing technologies. A new system of software technology totally redesigned for parallel processing, including algorithms, programming languages and operating systems, has to be established.

As the basis of this new technology, a concurrent logic language with extensions to control program execution on parallel hardware was designed as the *kernel* language of the system. An operating system that provides a comfortable environment for parallel application software development was also designed and implemented in the kernel language. This paper gives an overview of the research and development in this area of the FGCS project.

In the following sections, the design principles are described in section 2, the design of the kernel language in section 3, that of the operating system in section4. Experiences with the language and the operating system are described in section 5. Direction of future work is suggested in section 6, followed by concluding remarks.

2 Principles

2.1 Middle-Out Approach

When designing a computer system, two extreme approaches can be considered. One is a top-down approach, starting from problems to solve, gradually designing downwards to the level of computer architecture or even to the level of electronic devices, seeking in each level for a design most appropriate to implement higher levels. The other is a bottom-up approach, starting from

available device technologies, seeking for the best use of the lower level technology, finally finding an appropriate application area.

Neither of the approaches, however, cannot be successful by itself. In the top-down approach, design in each level requires insight into appropriate implementation of all the lower level technologies. In the bottom-up approach, design in each level requires insight into upper levels, up to application areas appropriate for the chosen design.

It is too difficult for anybody to have such insight for the broad and rather vague target of a long-term project, knowledge information processing. We thus decided to take a *middle-out* approach of designing a certain intermediate level first and conduct research and development towards two directions, upwards and downwards, simultaneously. It is not easy, of course, to find an appropriate intermediate level and to actually design that level. This, however, seemed to be the only feasible approach for a project like this one.

2.2 Kernel Language

The intermediate level we chose was the level of programming languages. Choosing this level has the following merits.

- The programming language level is not too far away from the both extreme ends of application software and hardware implementation.
- Relatively rigorous specification in the programming language level can be given more easily than in other levels.

The programming language designed to be the starting point of this middle-out approach is called the *kernel language* [Ueda and Chikayama 1990].

At the time the project started in 1982, language design and implementation technology was still premature to fix the design of the kernel language. Thus, the research started by investigating sequential systems first. In the first stage (fiscal years of 1982–84) of the project, a sequential kernel language based on Prolog, named ESP [Chikayama 1984], was designed, which formed the basis of the research and development in most of the research efforts in the first stage and early in the intermediate stage.

Design of the next version of the kernel language KL1 was started in the first stage simultaneously. Its preliminary design and implementation were done early in the intermediate stage and a fuller implementation on a experimental parallel computer system was completed within the intermediate stage (1985–88). The language has been used through the final stage (1989–) for various application research. In what follows, the kernel language means this second generation kernel language, KL1.

2.3 Logic Programming Principle

The logic programming idea gave the basis of the whole project. The image of logic programming in the original project plan seems to have been strongly influenced by a particular language Prolog. As the research proceeded from sequential systems to parallel systems, we had chosen a concurrent logic programming approach. The principle of placing "logic" as the central design principle, however, has been kept unchanged.

The principle of logic programming played a important role in selecting a particular design among many candidates. In designing the kernel language, its *soundness* in the sense of mathematical logic has been acted as a "canon", although we gave up pursuing *completeness*.[1] Many proposals to extend the kernel language with attractive features were investigated but rejected because of their *unsoundness*. On the other hand, features which do not change the meaning of the programs when interpreted as logical formulas were more freely added to the language. They have only to do with execution efficiency and nothing to do with the correctness of programs, and were clearly discriminated from the core part of the language.

These principles based on logical interpretation of programs have been quite helpful in keeping the language design coherent and, in its consequence, its implementation and its programming style coherent, as is described further in detail below.

2.4 Target Architecture

A processor with performance comparable to a full-size computer with reasonable amount of memory is now available on a single circuit board. Recent evolution of the hardware technology shows four-times increase in density of circuitry every three years. Extrapolating this, one hundred processors with reasonable amount of memory are expected to reside in one chip early in the next century. On the other hand, although the performance of single processor is steadily being improved, it might be very difficult to attain improvement by two orders of magnitude within the same time period.

With larger circuitry made practical with higher density, the design cost is beginning to dominate the total cost of processors. The design repeatability in multiprocessor systems will have great cost advantage over a complicated processor occupying one whole chip or more, even if the both systems had the same performance. Early in the next century, multiprocessor systems will thus be advantageous, not only in absolute processing power, but also in cost effectiveness even in small systems such as palm-top or wrist watch type computers.

[1] *Soundness* of a system means that any results obtained are logical consequences of the given axiom set. *Completeness*, on the other hand, means that all logical consequences can be obtained.

For application areas such as knowledge information processing that need non-uniform computation, an architecture that allows flexible resource allocation is required. For highly parallel systems, scalability of the system architecture is critical. Having these in mind, we chose a homogeneous MIMD architecture with loosely-coupled processors (or loosely-coupled *clusters* each with several tightly coupled processors) as the target architecture of the software system.

2.5 Level of the Kernel Language

An ideal programming language should allow very high level description with an implementation optimizing it to the target architecture without any human help. However, with the current technology, such a language is nothing more than a dream. It is especially so when the programs have to be optimized for execution on a large-scale loosely-coupled parallel computer systems where communication delay is not negligible. The most difficult part in the optimization will be *where* (on which processor) to execute certain parts of computation and *when* (in which order). Such a problem is known as the *mapping* problem.

As long as problem solving techniques used are relatively simple, required computation can be easily told beforehand making static mapping by compilers feasible. For knowledge information processing requiring sophisticated problem solving methods, what to compute next often depends on the result of the former steps of the computation, making static optimization of computation mapping impossible. Many research results have shown that general-purpose automatic mapping algorithm is hard to design and the selection of good mapping algorithms depends heavily on the problem solving method used.

As knowledge information processing is an area where no single universal and efficient problem solving method is known, providing one single mapping algorithm is not appropriate. Providing many mapping algorithms that cover all the known methods may still be insufficient; as research in the area is still in an early stage, many novel problem solving methods are expected to be proposed in the near future. Thus, we set the level of the kernel language so that mapping of computation can be specified in programs.

This decision of putting the responsibility of computation mapping on programmers has the drawback of making programming a more complicated task. We, however, regard this additional effort as unavoidable and essential in establishing the technology for high performance knowledge information systems. When a widely applicable mapping algorithm is established, it can be provided to the application users as a program library. With the kernel language capable of controlling program execution, writing such a library should not be difficult.

2.6 Designing a New Language

It might have been possible to take an already existing logic programming language as the basis of the kernel language and extend it with several additional features for concurrent execution. The logic programming language used most widely was (and still is) Prolog, which was the primary candidate for such extensions.

There could be two ways to tailor Prolog to a language for parallel systems. One method was to provide implicit and automatic computation mapping, which was not taken by the above-described reason. Another possible way was to make concurrent execution explicitly specified with additional language constructs. However, as the base language Prolog was designed for sequential processing, concurrency specification would add some more complexity to the language and making programs harder to understand. More importantly, if sequential execution should have made the default principle, it would have been more difficult to reorganize programs for better mapping, as different mappings require different parts of programs to run concurrently.

Another problem with such a language was pains in specifying synchronization. In programming languages in which synchronization is specified independent from conditioning, problems arise when decisions on conditional execution are made on incomplete data. On physically parallel hardware, finding such problems would become very painful because the same phenomenon is often hard to reproduce. To solve this problem, synchronization and conditioning should not be made separate.

We decided that the kernel language should be designed from scratch so that concurrent execution could be expressed in a natural way. The language should have intrinsic concurrency: language constructs imply concurrent execution in principle and sequencing is explicitly described. Synchronization should be integrated with conditioning in the language construct.

2.7 Designing a New OperatingSystem

Even though the prototype parallel inference system is an experimental system, an operating system that provides a comfortable software development environment was mandatory. One way to provide the required functionality might have been to port an already existing operating system to the parallel inference machine.

All the operating systems available then (and probably most of them even now) were designed originally for sequential systems and augmented afterwards with certain primitives for execution on parallel systems.

There were two major problems with such systems. One was that the interface of the operating system with the user programs was still based on sequencing. For example, the user program is notified of completion of requested service by the completion of execution of a pro-

cedure, supervisor call, in the user's thread of execution. This is acceptable in systems where application software is written in basically sequential languages. This, however, would not go well with software written in the kernel language with intrinsic concurrency.

Another problem was that the management policies of such operating systems were highly optimized for sequential processing. In sequential systems or small-scale parallel systems, centralization of all the management information is usually the most robust and efficient policy. This, however, is far from optimal for highly parallel systems. If the management were centralized on one processor in a highly parallel system, that processor would be responsible for too much management work and would be the bottleneck of the whole system. Moreover, every activity within the system would require communication to and from that processor, resulting in communication bottleneck.

We concluded that designing an operating system optimized for highly parallel systems was also an unavoidable and essential part of the technology for high performance knowledge information systems and decided to design and implement a new operating system from scratch. The user interface should be consistent with the design of the kernel language; sequencing should not be a part of the design of the interface. Distribution of management was essential to avoid bottlenecks, which might also affect the specification of the services provided by the operating system.

3 Kernel Language: KL1[2]

The kernel language KL1 has two layers. The basic layer is defined by Guarded Horn Clauses (GHC), which is a concurrent logic language for describing *what* computation to perform for desired result, that is, for describing *correct* programs. The description lays only those constraints on mapping of computation which are required to obtain the desired result. Based upon this layer is the full KL1 language for describing *how* such computation should actually be carried out with desired mapping of computation, that is, for describing *efficient* programs. This separation of correctness and efficiency issues or, in other words, concurrency and parallelism, seems to play an important role in bridging the gap between parallel inference systems and knowledge information processing in a coherent manner.

3.1 Concurrent Logic Language GHC

This section describes the design of a concurrent logic language Guarded Horn Clauses, which forms the basis of the kernel language KL1.

[2]This section is a rewrite of an article co-authored with Kazunori Ueda [Ueda and Chikayama 1990], except for the subsection 3.3.

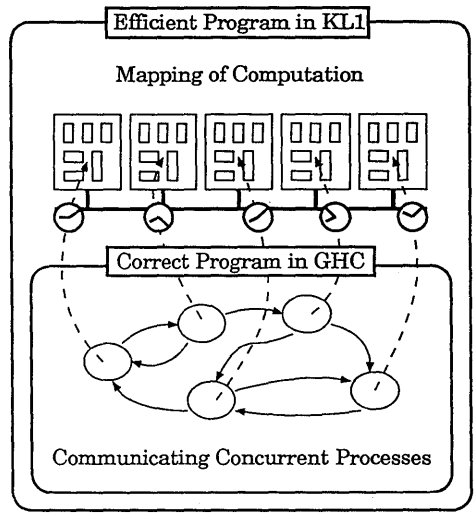


Figure 1: Two Layers of the Kernel Language

3.1.1 Concurrent Logic Languages

The design effort of the kernel language was started in 1982 with the start of the project by seeking for an appropriate framework of the language. As the concurrent logic programming framework seemed to provide the characteristics in our need, we investigated many languages in the family as the basis of the kernel language, including Relational Language [Clark and Gregory 1981], Concurrent Prolog [Shapiro 1983] and PARLOG [Clark and Gregory 1983]. This study led us to a design of a new concurrent logic language, Guarded Horn Clauses (GHC) at the end of 1984 [Ueda 1986].

GHC shares its basic framework with other concurrent logic languages. Firstly, a GHC program is a set of *guarded* clauses. Secondly, GHC features no don't-know nondeterminism (built-in search capability) but features don't-care nondeterminism, which allows description of reactive systems. Reactive systems in concurrent logic languages are based on the process interpretation of logic [van Emden and de Lucena Filho 1982], in which a goal (or a multiset of subgoals derived from it) is regarded as a process and processes communicate by generating and observing bindings (between shared logical variables and their values). Like most concurrent logic languages, all bindings are *determinate* in GHC, that is, they are never revoked once published to other processes. The determinacy of bindings is essential in reactive systems, such as an operating system, because the bindings may be used for interacting with the real outside world. The lack of built-in search capability also allows programs to specify the way of their execution in more detail, which

ılso matches our principle of making programs specify napping of computation.

3.1.2 Guarded Horn Clauses

What then is the relative merit of GHC over other concurrent logic languages? In our study of various concurrent logic languages, we focused on Concurrent Prolog, which was the most expressive of them, and built its prototype implementation [Miyazaki *et al.* 1985]. The experience led us to clarify the definition of atomic operations of the language, which in turn led us to a new language with simpler atomic operations.

As explained above, one important aspect of concurrent logic languages is the determinacy of bindings. In general, the execution of a concurrent logic program proceeds using parallel input resolution [Ueda 1988a] that allows parallel execution of different goals, but under the following rules to guarantee the determinacy of bindings:

(1) The guards (including the heads) of different clauses called by a goal g can be executed concurrently, but they cannot instantiate g.

(2) The goal g *commits* to one of the clauses whose guards have succeeded.

(3) The body of a clause to which g has committed can instantiate g. The bodies of clauses to which g has *not* committed cannot instantiate g or the guards of the clauses.

(4) A goal is said to *succeed* if it commits to some clause and all its body goals succeed.

That is, before commitment, a goal can pursue two or more clauses but without generating bindings. After commitment, it can generate bindings but only one clause is left.

Another important aspect of concurrent logic languages is how synchronization is achieved. In general, synchronization is achieved by restricting information flow caused by unification. Concurrent Prolog uses read-only annotations, and PARLOG uses mode declarations which are used for compiling the unification of input arguments into a sequence of one-way unification and test unification primitives. However, in these languages, additional mechanisms are necessary to guarantee restriction (1) above.

The key idea of GHC is quite simple. It uses the restriction (1) itself as a synchronization construct. That is, any piece of unification which is invoked directly or indirectly from the guard of a clause C and which would instantiate the caller of C is suspended until it can be executed without instantiating the caller. In other words, GHC has integrated two notions: the determinacy of bindings and synchronization.

A kernel language must provide a common framework for people working on various aspects of the project including applications, implementation, and theory. Before accepting GHC as the basis of our kernel language, we had to convince ourselves that it satisfies the following conditions:

- It is expressive enough.
- It can eventually be implemented efficiently, possibly by appropriate subsetting.
- It is simple enough to be understood and used by programmers. Also, it is simple enough for theoretical treatment.

We soon made sure that GHC was expressive enough to write most concurrent algorithms that had been written in other concurrent logic languages, but that was not enough. How to program search problems was also important, because search problems are a specialty of ordinary logic languages. So we have developed a couple of methods for programming search problems [Ueda 1987], [Tamaki 1987], [Okumura and Matsumoto 1987].

For implementability, we quickly ascertained by rapid prototyping that GHC can be implemented fairly efficiently at least on sequential computers [Ueda and Chikayama 1985].

3.1.3 Flat GHC

For simplicity, we continued to study the properties of GHC and looked for a simpler explanation of the language better suited to process interpretation. Now, our interpretation is that a GHC process is an abstract entity which observes and generates information (represented in the form of bindings) and which is implemented by a multiset of body goals. The behavior of each body goal is defined by guarded clauses that can be regarded as rewrite rules.

A problem with the original definition of GHC is that guard goals do not fit well into this process interpretation. We also felt, from a practical point of view, that the expressive power of guard goals did not justify the implementation effort even if it could be implemented efficiently.

These considerations led us to reduce GHC to a subset, Flat GHC. Guard goals of Flat GHC are auxiliary conditions to be satisfied for applying the clause. The sufficient conditions to be satisfied by a guard goal as an auxiliary condition are that it is deterministic (that is, whether it succeeds or not depends only on its arguments) and that it does not produce any bindings. This restriction simplified the theoretical treatment considerably in the operational semantics [Ueda 1990] and program transformation rules [Ueda and Furukawa 1988].

To summarize, a Flat GHC program is a set of guarded clauses that can be regarded as rewrite rules of goals.

The guard of a clause specifies what information should be observed before applying the rewrite rule, and the body specifies the multiset of goals replacing the original. A body goal is either a unification goal of the form $t_1 = t_2$, whose behavior is language-defined, or a non-unification goal, whose behavior is user-defined. A unification body goal generates information by unifying t_1 and t_2, and a non-unification body goal represents the rest of the work and will be reduced further.

3.1.4 Characteristics of GHC

The semantics of Flat GHC can be understood both algebraically and logically. The algebraic one is the process interpretation mentioned above. A logical characterization of communication and synchronization was given by Maher [Maher 1987], showing that information communicated by processes can be viewed as equality constraints over terms.

Unlike Concurrent Prolog but like PARLOG, the publication of bindings is not done atomically *upon* commitment of a non-unification goal but eventually *after* commitment using a unification body goal that can run in parallel with other goals. This means that commitment in GHC is a smaller and simpler operation than in Concurrent Prolog. Moreover, in GHC, the information generated by a unification body goal is not an atomic entity but can be transmitted in smaller pieces, possibly with communication delay. We have found that this liberal computational model of (Flat) GHC is expressive enough to program cooperating concurrent processes and leaves more freedom to implementation.

Another point to note is that GHC has included control for the *correct* behavior of processes but excluded any control for *efficient* execution. GHC has left the latter to KL1 described below, in order to clearly distinguish between the two notions. This contrasts with PARLOG, which features sequential AND that can be used for suppressing parallel execution of body goals. We believe that it is important to learn that synchronization based on information flow is sufficient for writing correct concurrent programs.

Important topics on theoretical aspects of Flat GHC include the relationship with other theoretical models of concurrency such as CCS [Milner 1989] and theoretical CSP [Hoare 1985]. Although concurrent logic languages differ from CCS and CSP in their asynchronous communication and dynamically reconfigurable processes, similar mathematical techniques can be used to formalize them. We have not yet obtained a completely satisfactory formal semantics, but we are fairly confident that Flat GHC is theoretically simple enough, while it can be used for practical programming without any modification.

3.2 Practical Parallel Language KL1

As described above, we have designed a concurrent logic language Flat GHC as the basis of the kernel language. The descriptive power of the language, however, is not sufficient when efficient program execution is our concern, which was the original motivation of parallel computers.

As Flat GHC programs do not say anything about where (i.e., on which processor) the atomic operations making up a computation should be performed, there are many ways to distribute the operations over available processors. As Flat GHC programs only specify the partial ordering of atomic operations, there are many possible total orderings conforming to it. To make sure that the distribution and the ordering employed are not far from optimal, we must be able to specify physical details of execution to some extent.

We thus designed a parallel programming language based on the concurrent programming language Flat GHC, in which we can specify in certain detail *how* a program should be executed. This section describes the outline of this language, named KL1.

3.2.1 Mapping of Computation

Flat GHC programs implicitly express any potential parallelism in the sense that no ordering between atomic operations exists except for those essential for correctness. On real-world computer systems with a limited number of processors and non-negligible cost of interprocessor communication, faithful exploitation of this parallelism will almost never show optimal efficiency. To achieve reasonable efficiency, control is required on when and where each atomic operation should be performed. This control is called *mapping*.

Mapping is often implicit in sequential systems. With two possible methods to solve a problem, a good strategy on a sequential system would be trying more efficient but less reliable one first and trying less efficient but reliable one second only when the first one fails. This may not be the best for parallel systems, when the first method will not require all the computational resource (such as processors) for its execution. In such a case, the second method should be tried in parallel with the first. This computation may or may not be required depending on the result of the first method. Such computation is called *speculative* [Burton 1985]. For efficiency, computation by the second method should not interfere the execution of the first by snatching required resources. This is effected by giving priority to the first method over the second. From this viewpoint, the original sequential algorithm uses sequencing of two methods not for correctness but for efficiency to implicitly specify priority.

Sometimes more sophisticated mapping is desirable. Suppose that there are two methods to solve a problem and that, although at least one is known to find a so-

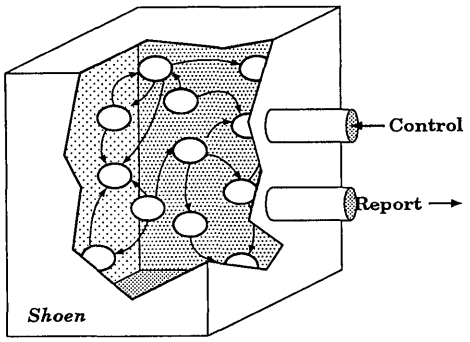


Figure 2: *Shoen* Construct

lution efficiently, we cannot tell which beforehand. In such a case, the best scheduling strategy may be to give both methods approximately the same amount of computational resource. Resource management is thus an important part of an algorithm in parallel computation.

In sequential computer systems and in parallel computer systems as extensions of conventional sequential systems, operating systems are primarily responsible for mapping. This is acceptable as far as application programs are mostly sequential and the mapping strategy is often specified by sequencing implicitly. In parallel systems where explicit mapping operations are much more frequently required, requesting each mapping operation to the operating system would incur intolerable overhead.

3.2.2 Mapping Features of KL1

To solve this problem, we have introduced into KL1 the following features, which are intended to be efficiently implemented:

Shoen: *Shoen*[3] represents a group of goals. This group is used as the unit of execution control, namely the initiation, the interruption, the resumption and the abortion of execution. Exception handling and resource consumption control mechanism are also provided through this *shoen* construct. It has two communication streams as its interface: one directs from outside of the *shoen*, called *control stream*, for sending messages to control execution in the *shoen*; the other, called *report stream*, has the reverse direction for reporting events internal to *shoen*. The *shoen* construct is an extension of the *metacall* construct proposed by Clark and Gregory [Clark and Gregory 1984].

Priority: A (body) goal of a KL1 program is the unit of priority control. Each goal has an integer priority associated with it. Each *shoen* keeps the maximum and the minimum priorities allowed for goals belonging to it, and the priority of each goal is specified relative to these. The language provides a large number of logical priority levels, which are translated to physically available priority levels provided by each implementation.

Processor specification: Each (body) goal may have a processor specification, which designates the processor (or a group of processors) on which to execute the goal.

This straightforward mechanism provides the basis of research in more sophisticated computation mapping strategies. Actually, several automatic mapping strategies have been developed for diverse problems, and relatively universal ones are provided as libraries [Furuichi *et al.* 1990].

One of the most notable characteristics of the KL1 language is that these priority and processor specifications are separated from concurrency control. We call these specifications *pragmas*. Pragmas are merely guidelines for language implementations and may not be precisely obeyed. The same is true of the controlling mechanism of *shoen;* abortion of computation, for example, may not happen immediately. This relaxation makes distributed implementation much easier.

In many parallel programming languages, the specification of parallel execution is often mixed up with other language constructs, especially with constructs for concurrency control. A major revision is often required for revising only the mapping of computation to improve efficiency, which is liable to introduce new bugs.

Although pragmas are specified within the program in KL1, they are clearly distinguished syntactically from other language constructs. Pragmas will never change the correctness of the programs,[4] though the performance may change drastically. As it is not uncommon that more than half of the effort to develop a program is devoted to the design of appropriate mapping, it is most advantageous that mapping specifications can be altered without affecting correctness of the program.

3.2.3 Keeping up with Sequential Languages

What criterion is appropriate for comparing parallel algorithms? Assume that a parallel algorithm has sequential execution time $c(n)$ (n being the size of the problem) and average potential parallelism $p(n)$. Then the total execution time by this algorithm on an ideal parallel computer is given by $c(n)/p(n)$. This means that an algorithm with more sequential execution time but with still more parallelism is considered to be a better algorithm on an ideal parallel computer.

[3] *Shoen* is a Japanese word corresponding to 'manor' in English.

[4] To be precise, the priority specification may be used for guaranteeing certain properties of diverging (i.e., autonomously nonterminating) programs.

This, however, does not hold when the potential parallelism, which may vary over time, can exceed the physically available parallelism. As physical parallelism is always limited in the real world, a parallel algorithm with sequential time complexity worse than a sequential algorithm will be beaten by that sequential algorithm for sufficiently large n, *no matter what* $p(n)$ *is*. To summarize, parallel languages must be able to express any algorithms with the same sequential time complexity as in sequential languages to be really useful.

Pure languages such as pure Lisp and pure Prolog cannot express certain kinds of efficient algorithm due to the lack of the notion of destructive assignment. GHC also is a pure language with the same inherent problem. To write efficient algorithms in these pure languages, we must be able to somehow mimic the efficiency of array operations in conventional languages.

For this reason, KL1 introduced a primitive for updating an array element in constant time without disturbing the single-assignment property of logical variables. The primitive can be used as follows:

```
set_vector_element(Vect, Index,
                   Elem, NewElem, NewVect)
```

When an array `Vect`, an index value `Index` and a new element value `NewElem` are given, the predicate binds `Elem` to the value of the `Index`'th element of `Vect`, and `NewVect` to a new array which is the same as `Vect` except that the `Index`'th element is replaced by `NewElem`.

Because some other goals may still have references to the old array `Vect`, a naive implementation might allocate a completely new array for `NewVect` and copy all but one elements. However, when it is known that no goals other than the above `set_vector_element` goal have references to `Vect`, there will be no problem in destructively updating it. In the actual implementation of KL1, a simplified, efficient version of the reference counting scheme [Chikayama and Kimura 1987] detects such a situation, in which event the new array `NewVect` is obtained in constant time.

This means that any imperative sequential algorithm can be rewritten in KL1 retaining the same computational complexity, as random access memory can always be emulated using a single-reference array. Of course, allowing only one reference to a data structure can decrease the possibility of parallel execution considerably. However, this requirement of the computational complexity becomes essential only after physically available parallelism is used up.

3.3 Higher-Level Languages

Although the kernel language KL1 allows relatively higher level description of programs than imperative languages, its description level is in the same level as Lisp, which is still too low for certain application programs in the area of knowledge information processing. This section describes research on providing higher-level language constructs upon KL1.

3.3.1 Macro Expansion

A powerful macro expansion mechanism similar to the one available in ESP [Kondoh and Chikayama 1988] is designed and implemented. This macro allows not only in-place expansions of macro invocations but also insertion of terms into the program in the levels of arguments, goals or clauses. The following are possible using these features.

- Simple in-place expansion
- Conditional compilation
- Functional notations including but not restricted to arithmetical expressions
- Implicit arguments

A goal of Flat GHC programs has very short lifetime, as it consists of only one reduction to its subgoals. To realize a process with longer lifetime, a programming style is used in which a goal recursively calls the same predicate with almost the same arguments. This programming style is used almost everywhere in the operating system and application programs. In such a programming style, the state of the process or any paths to communicate with other processes (shared variables) have to be passed as the arguments of the recursive goal. This ensures higher modularity, but always describing such arguments is too verbose, making it harder to understand or to revise programs. The implicit argument passing mechanism can be conveniently used to describe processes in a more concise manner.

The macro expansion mechanism of KL1 is so powerful that functions beyond mere syntactic sugaring can be provided using its features. However, programmers can freely choose any programming style allowed in KL1. Although this is advantageous in certain cases, restriction on the usage of the language features is profitable in making programs easier to understand and maintain. We thus started designs of higher-level languages to be compiled into KL1, which will be described in the following sections.

3.3.2 A'UM

The programming style of KL1 most frequently used is to describe a set of processes communicating through message streams [Shapiro and Takeuchi 1983]. Streams are realized by gradually instantiating a list structure consisting of binary cells. Processes are realized using tail recursion. A'UM is a programming language designed to describe such programs more directly than explicitly

writing such realization of message streams and processes [Yoshida and Chikayama 1990].

A prototype implementation of the language was a translator to KL1. As a thoroughly object-oriented language, every entity of the language A'UM, an integer value for example, appears as a process. We could find no other way than to actually implement them as processes in KL1. The choice then was whether to abandon thorough object-orientationor to implement it differently, not as a part of the parallel inference system. A'UM took the latter choice and research on its more direct implementation is ongoing [Konishi *et al.* 1992]. A prototype implementation is already operational on a system of network-connected workstations. The former approach was taken by another language with similar objectives, called AYA, which is described in the next section.

3.3.3 AYA

The design of the language AYA was initiated after we decided to let A'UM seek for pure object-orientation rather than pursue practical efficiency on the parallel inference system [Susaki and Chikayama 1991].

The design objective of AYA is the same as the initial motivation to design A'UM, namely, providing a more concise way to describe programs in object-oriented programming style of KL1. In design of AYA, a higher priority is given to practical efficiency and freedom of description than uniformity as an object-oriented languages. Not all entities are "objects": integers will not respond to "add" messages. Its design was mostly bottom-up; most of the language features were chosen based on our programming experiences in KL1.

Processes of AYA can have multiple streams to receive messages, making it impossible to interpret one single message stream to be representing an object. Communication patterns besides streams such as asynchronous interrupts are also allwoed.

A characteristic feature of AYA is the notion of *scenes*, corresponding to the macroscopic context of a process. A process can have many scenes to act in and its reaction to messages from outside will depend on in which scene it is currently acting.

Implementation effort of AYA is ongoing and a prototype translator to KL1 is already operational.

4 Operating System: PIMOS

As described above, an operating system tuned to control highly parallel programs effectively is vital for fully exploiting the power of highly parallel computer systems. The system should also be user-friendly and robust enough for practical and extensive use in parallel software research. The Parallel Inference Machine Operating System (PIMOS) was designed to fulfill the requirements and implemented in the kernel language. This section describes the overall design of PIMOS.

4.1 Prior Works

The possibility and advantages of writing a complete operating system in a concurrent logic language were suggested by Shapiro [Shapiro 1986]. Based on this principle but with much improvements in various aspects, several experimental systems such as the Logix system [Hircsh *et al.* 1987] and the Parlog Programming System (PPS) [Foster 1987] were implemented.

PIMOS resembles PPS in many aspects. This resemblance is partly due to the resemblance of the implementation languages (KL1 and PARLOG) and partly due to frequent exchange of ideas among the two groups.

A notable difference between PIMOS and the other above-mentioned systems lies in the underlying language implementations and the way the system is used. PIMOS is designed to be efficiently executed on a parallel hardware to be practically used in the research and development of application software, while other systems are built as experimental systems upon commercially available systems. In other words, PIMOS shares with other systems the objective of seeking for a novel method of constructing an operating system in concurrent logic language, but has an additional objective of providing a comfortable and efficient environment for application software development. This considerably affected various design trade-offs.

4.2 Objectives

In designing PIMOS, the following items were set as the design objectives.

Robustness: As PIMOS is to be used on a stand-alone parallel computer system, the robustness of the system is more important than in systems build upon another established system.

Internal Parallelism: The ultimate objective of PIMOS is, as stated above, to provide features for fully exploiting the power of parallel inference hardware. Various computation required in such an operating system should also be executed in parallel. Otherwise, the operating system will be the bottleneck of the whole system.

High Locality: The target architecture has loosely-coupled processors where inter-processor communication is much more costly compared with communication within one processor. Thus, the amount of communication between processors should be kept as low as possible.

Flexibility: As the hardware parameters are expected to change, the system should have enough flexibility

to be tuned to the given parameters. When tuning by changing parameters of the operating system becomes insufficient, non-trivial re-design of the system may be required. Thus, a system on whose improvement is easy is desirable.

4.3 Resource Management

Management of resources is the most fundamental and important role of an operating system. This section describes the design of the resource management mechanism of PIMOS.[5]

4.3.1 What Resources to Manage

In conventional systems, memory management and process management are the most important tasks of operating systems. As in other high-level language for symbolic manipulation, KL1 provides an automatic memory management feature including garbage collection. Thus, basic memory management is by the language implementation rather than PIMOS. As KL1 provides implicit concurrency and data-flow synchronization, context switching and scheduling are already supported by the language. Thus, PIMOS does not have to manage low-level fine-grained processes, but controls larger-grained groups of processes using the *shoen* feature of the kernel language.

On the other hand, PIMOS has full responsibility on the management of resources such as input and output devices. In the lowest level, I/O devices are provided as primitives of the kernel language to control physical device interface. Thanks to the descriptive power of the kernel language for reactive systems, such devices have a disguise of an ordinary process in the kernel language level. Their functionality, however, is at a level too low for application programs. Like any other operating systems, PIMOS virtualizes such devices, allowing application programs to control virtual devices with much higher-level functionality.

These virtual devices are actually a process that converts higher-level requests from user tasks into lower-level requests that physical devices can understand. The user tasks send their request messages to a stream connected to such a process. Thus, management of devices is management of the communication streams connected to them. Protection mechanisms are realized by inserting a filtering process to such streams, which examines messages going through the stream and rejects any illegal requests to the devices.

As mentioned above, process management by PIMOS is through the *shoen* construct. PIMOS virtualizes *shoen* also as a *task* with higher-level functionality for resource management. Tasks are a virtual device with the function of program execution with resource management facility. They can be controlled from user programs only through streams connected to it. The same protection mechanism of inserting message filtering processes is used here.

[5]More detailed description can be found in [Yashiro *et al.* 1992].

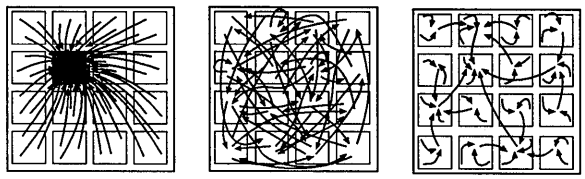


Figure 3: Distribution of Management Jobs

4.3.2 Hierarchical Resource Management

In most conventional operating systems, all the vital management information is centralized to the kernel, which is usually implemented as a single process. This centralization policy makes it easy to keep the management information consistent.

In a highly parallel system, however, such centralization of management information would become problematic. Even if the overhead of the kernel is only one percent, the processing speed of the kernel will be the bottleneck of the system in a system with only one hundred processors. Moreover, all the management requests will be targeted to the processor where the kernel process runs, resulting in a hot spot in the communication mechanism. In an operating system for highly parallel computer systems, management jobs also have to be distributed.

Random distribution of management jobs, using hashing technique for example, would relieve the bottleneck problem, but introduces a new problem of frequent communication, as the requests for operating system services arise everywhere without regard to where the service is provided.

To avoid the bottleneck and frequent communication at the same time, it is essential to distribute management jobs keeping the locality of information. PIMOS, thus, adopted hierarchical resource management policy. User tasks and resources allocated by the operating system form a hierarchical structure. As the design principle leaves computation mapping to application programs, processes of PIMOS responsible for management jobs will be allocated where requests for services arise, and those management processes also form a hierarchical structure corresponding to the structure of user tasks, called *resource tree.* This resource tree is the *kernel* of PIMOS.

No centralization of resource management information is made and no total ordering of resource allocation is

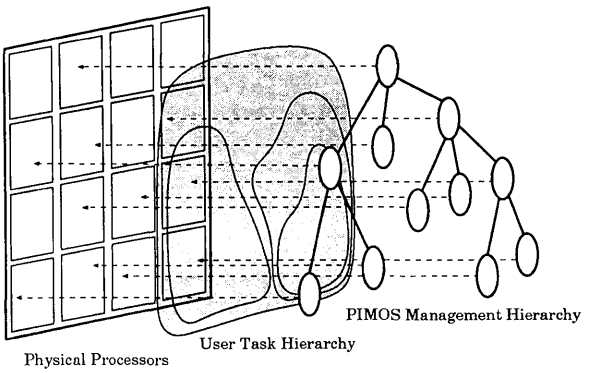


Figure 4: Task and Management Hierarchies

tried. A management process, which is a node in the resource tree, knows only of its parent and children. Allocation of a new resource is handled locally at one level in the hierarchy without reporting it to upper levels nor lower levels. When necessary, statistical summaries of management information is exchanged in the resource tree, but there is no single process that knows the state of the whole system precisely. The state of the whole system can be investigated by traversing the tree structure, but that would be costly and, because of the concurrent activities in the system, obtained information might already be obsolete when the the traversal completes. We found this loose management policy works fine without any problems.

4.3.3 Servers

All the services of PIMOS are provided by *servers,* which correspond to virtualized devices. Servers are realized as usual tasks to make the kernel compact and to enable easy addition of services.

An application program (client) requiring a service (to open a display window, for example) can ask for the service by requesting to the kernel with the name of the service. The kernel will look for the named service in a table it maintains and establishes a stream connection between the server task and the client task, inserting a filtering process for protection in the client task at the same time. Once the connection is established, the kernel will not look into messages passed through the stream; the server is protected by the inserted filter rather than a kernel process. When the service become no longer needed, the client process normally closes the communication stream. The remaining responsibility of the kernel is to notify the server of abnormal termination of the client.

4.4 File System

Earlier versions of PIMOS operating on an experimental model Multi-PSI [Takeda *et al.* 1990] left all the external input and output to its I/O front-end processor, PSI [Nakashima 1987]. This was profitable in rapidly constructing a software development environment for applications research. For massive external storage, such as disks, the imbalance of the low throughput communication with the I/O front-end and high performance processing power of the parallel hardware, however, became more apparent with PIM [Taki 1992].

We thus decided to connect disks more directly to processors of PIM for higher throughput and shorter delay. To minimize hardware development effort, we adopted SCSI (small computer standard interface) to interface disks available in the market. Although single SCSI can provide rather low throughput, PIM can have many of them, providing required total throughput.

As the interface provides only low-level block I/O to disks, we designed a file system to provide higher-level interface to application programs. In designing the file system, we took the following principles.

Distributed Cache: To lower interprocessor communication frequency, each processor should have its own cache of data in file. The cache mechanism should provide "Unix semantics": When one process writes into a file, the data should become available to other processes *immediately.* This is a constraint severer than in many distributed file systems where some delay is allowed [Levy and Silverschatz 1989], but it is mandatory in a system like PIMOS, where processes are usually cooperatively solving one problem. Thus, a distributed and coherent caching mechanism was designed, which is similar to cache coherence mechanisms provided by snoopy cache [Archibald and Bare 1986] but allows delay of communication.

Robustness: As all the system components, including the hardware, the operating system and the file system itself, are experimental and subject to damage caused by bugs, sufficient backing up mechanism is required to provide a comfortable software development environment. Logging of information vital to the file system and quick recovery mechanism using the logged information were designed.

More detailed description of the file system can be found in [Itoh *et al.* 1992].

4.5 Software Development Tools

Development of parallel software has many aspects different from development of sequential software. PIMOS provides various tools to support development of parallel software, described in this section.

4.5.1 Program Code Management

Executable programs are provided as data objects of type *module* by the kernel language and can be manipulated through language primitives by authorized software. Although the representation of executable programs differ in each hardware models, a common interface to manipulate programs is provided by PIMOS to encapsulate the differences.

Executable programs are stored in a database, which is a virtual device realized by a server task. To maintain the logical soundness of the specification, it is not desirable to introduce the notion of *modification*, not only for usual data but also for programs which are also data. Updating a program module does *not* mean modification of an already existing program, which might be running in parallel somewhere in the system; it merely means updating of the correspondence of module names and executable programs kept in the program database. The existing processes that are executing the program will not be affected by this update, except that, when the updated module is referenced by its name and the database is searched for, a new version of it will be found. Multiple versions of the same program can thus coexist in a system. This not only keeps the semantics clean but also allows efficient distributed implementation.

4.5.2 Debugging Tracer

The most frequently used tools in debugging programs are tracers that allow programmers to look into the details of program execution. PIMOS also provides a program tracer for this debugging purpose.

Execution of programs in a high level language form a hierarchical structure such as nested subroutine calls. In case of subroutines in sequential languages, substructures corresponding to subroutine invocations directly correspond to a time interval, such as "during execution of a subroutine." Tracing or not tracing that particular substructure can be effected by switching tracing on and off during that time interval. In concurrent languages, such direct correspondence does not exist as many such substructures are executed concurrently. If the number of processes is limited, providing multiple windows, one for each process, and switching tracing on each of them might be a good idea. In case of KL1 programs, the number of processes typically goes up to millions, much more than tractable this way. The tracer of PIMOS also provides a feature to direct the trace information to multiple windows, but their role is only auxiliary.

The *shoen* construct of the kernel language is used to control tracing, to obtain trace information and to control execution of traced programs. Each goal executed in a *shoen* can be marked as a *traced* goal. When the language implementation finds reduction of such a goal to its subgoals, the newly created subgoals will be reported from the report stream of the *shoen* as a message. The tracer observing the stream presents the information to the user and queries what to do with the goals, that is, whether to simply execute them or execute them with *trace* marks again. The goals can also be suspended for a while to control their execution order.

The tracer also has interface with the deadlock detection mechanism provided by the KL1 implementation [Inamura and Onishi 1990].

4.5.3 Performance Tuning

As stated above, a strong point of the kernel language KL1 is that mapping of computation, both over processors and over time, can be altered without affecting the correctness of programs. Finding a mapping which realizes efficient computation is one of the most important research topics in application software research on the parallel inference system.

However, conjecturing mapping only by statically analyzing programs is a very difficult task. In many cases, actually running the programs and gathering statistical information reveals many aspects of programs that are easily overlooked. To help such experimentation, PIMOS provides a tool for evaluating load distribution algorithms.

Profiling information of parallel programs has three axes: what, when, and where. In sequential execution, "where" is a constant and the "when" is not important, since the execution order is strictly designated. Simple profiling tools that can tell "what" (which part of the program) took how much time will thus suffice. However, all three axes are important when parallel execution is our concern. The kernel language implementation has the feature to provide three-dimensional statistics on *what* (which part of the program, or, in a lower level, whether usual computation, interprocessor communication or garbage collection) is executed *where* (on which processor) and *when*.

As it is not easy for a human to understand massive raw data from hundreds of processors, a profiling tool named *ParaGraph* is provided to analyze the data and present it to the user graphically (Figure 5). The system provides displays from several different viewpoints, making the analysis easier. The ParaGraph system is described in more detail in [Aikawa 1992 *et al.*].

4.5.4 Virtual Machine

As all the communication between user programs and PIMOS is initiated through the control and report streams of *shoens*, a user program can emulate PIMOS by running programs within a *shoen* and observing its interface streams.

The same technique also can be used to debug PIMOS itself by writing an emulator of the whole parallel computer system, a *virtual machine*. This facility provides a way to debug PIMOS under the software environment

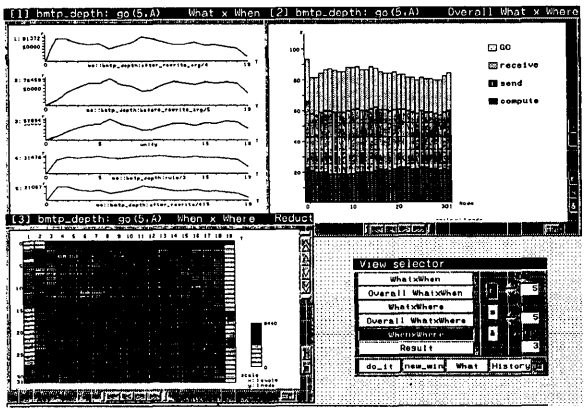


Figure 5: Sample Output of ParaGraph

provided by PIMOS itself. As the virtual machine is no more than a usual task in PIMOS, the protection mechanism of PIMOS prevents bugs of the debugged version from propagating to the real PIMOS. Also, the profiling system ParaGraph can be used for performance tuning of PIMOS. This facility has been conveniently used in debugging and tuning of the kernel of PIMOS.

5 Experiences

The first version of PIMOS was implemented on Multi-PSI [Takeda *et al.* 1990] in 1988 [Chikayama *et al.* 1988]. It has been revised with various enhancements and improvements since, through experiences with research and development of experimental software on many application areas. As the experiences with application software are reported elsewhere (see [Nitta *et al.* 1991] for example), this section mainly reports the experiences of the development of PIMOS itself in the kernel language KL1.

5.1 Automatic Synchronization

The automatic data-flow synchronization mechanism of KL1 assured portability of PIMOS to hardware systems with different architectures.

The first version of PIMOS was developed in parallel with the development of the experimental parallel inference machine Multi-PSI. During its early development phase when no physically parallel system running the kernel language was available yet, a sequential implementation was used in the development. The scheduling of goals was fixed on the implementation. We could not completely deny the possibility of any crucial synchronization problems in the system hidden by the fixed scheduling of the emulator; that was our first experience of actually writing a large-scale software in KL1.

PIMOS was ported to Multi-PSI when its KL1 implementation got ready. We found almost no synchronization problems there (except for a small number of higher-level design problems) although the scheduling on the real parallel machine is quite different from the emulator. We were certain that this should be the case, but actually experiencing this made us more confident of the great merit of writing a system in a language with automatic data-flow synchronization.

In 1991, the first model of the parallel inference machines, PIM/m and its KL1 implementation was made available for software installation. After revising the low-level I/O mechanism to fit the system to this new platform, PIMOS began working almost immediately on this system without revealing any problems. This was not surprizing as the kernel language implementation on the system used the identical scheduling policy as the Multi-PSI system.

Later in the same year, the system was ported to an emulator of PIM running on a commercially available parallel processor. The emulator was primarily for debugging the design of kernel language implementation for models consisting of loosely-coupled *clusters*, each of which has several processors sharing a memory bus. The scheduling policy of this emulator was completely different from Multi-PSI or PIM/m, as the language implementation distributes goals automatically among processors in a cluster. As we expected, and also to our surprise, PIMOS ran without any problems in itself but revealing some problems with the language implementation in stead.

Currently (February 1992), the kernel language implementation and PIMOS are being ported to other models of PIM. We are now certain that there won't be any fundamental problems in porting PIMOS to those models.

5.2 Fine-Grain Concurrency

It is true that most human algorithm designers are liable to regard computation as a sequential process and some extra effort is needed to think of many cooperating processes for a single job. This fact is sometimes regarded as against parallel processing, that designing parallel computation is unnatural for human. The implicit concurrency of the kernel language, however, resulted in interesting phenomena.

Most algorithms in fact are designed having sequential processing in mind or limited aspects of the parallelism. Once a program for the algorithm is written down in the kernel language, the program often shows much more concurrency than the designer had in mind, as the language reveals implicit fine-grain concurrency. The designer can look into the program more objectively and find different aspects of concurrency implied there. Sometimes, the concurrency so found is a good candidate for obtaining larger physical parallelism for increased ef-

ficiency. Mapping pragmas exploiting the concurrency can then be added to the program to make it run with higher parallelism and more efficiently. This should not have been possible if the language had only larger-grain concurrency.

5.3 Descriptive Power

Through the development of PIMOS, the descriptive power of KL1 for both concurrency and parallelism was proved to be sufficient.

The ability of describing reactive systems allowed the language to provide primitives to control external I/O devices in a coherent manner; external devices could be modeled as an ordinary process without introducing any extralogical features to the language. This allowed straightforward implementation of a virtual machine, which helped the development considerably.

The *shoen* construct and the priority control mechanism of the kernel language provided sufficient functionality required to control execution of various activities in the system. For example, in case a user program ran into an infinite loop, the following steps will enable interruption of such a program.

- As the device handlers are given higher priority than user processes, an interrupt from the keyboard can be sensed.
- As the command shell, which is a user task, lets jobs under its control run in a priority lower than itself, the shell can sense the interrupt.
- Using the *shoen* construct, the shell can stop the task in an infinite loop.

5.4 Ease of Programming

Many programmers seem to have felt uneasiness with the kernel language when the system first began utilized in application software development. The largest source of the problem seems to be in too much freedom of programming styles.

The bare kernel language allows multiple input/output modes of logical variables; the same process can read or write the same shared variable, depending on situations. Although this is allowed in the language, it often introduces race conditions which become problematic only with specific scheduling. Such a bug is hard to fix as tracing the execution or modifying the program to report information for debugging may change the scheduling, hiding the problem away. Gradually, a programming style has been established where I/O modes of logical variables are statically fixed. This indicated the direction of subsetting of the language (see section 6).

Another problem was how to organize numerous concurrent processes. Many styles have been tried and the object-oriented programming style [Shapiro and Takeuchi 1983] has been accepted as the *de facto* standard. Many programming idioms have been established upon this object-oriented style through experiences [Chikayama 1991], which suggested the direction of the design of higher level languages (see section 3.3).

Automatic data-flow synchronization wiped away low-level synchronization problems, allowing programmers to concentrate on higher-level issues. With the programming style established and the software development environment enhanced based on the experiences, describing parallel software in the kernel language has now become not much more difficult than programming sequential programs in other languages for symbolic processing, such as Lisp.

The largest difficulty remaining is that of designing algorithms of computation mapping for efficient execution. Separation of correctness and efficiency issues in the language design and the visual performance analysis tool facilitated experimentations of mapping algorithms considerably, but still the task is not easy. Further research in this direction seems mandatory.

6 Future Work

A problem with the current parallel inference system, consisting of parallel inference machines, KL1 implementations and PIMOS, is that the system runs only on specially devised hardware. Although the system can execute KL1 programs very efficiently, requiring special hardware is a serious obstacle in sharing the environment with researchers world-wide. A portable implementation of the kernel language working on Unix systems is available and was utilized in early stages of software development, but, as it is implemented as an abstract machine interpreter, its limited performance makes it inappropriate for serious experimental studies.

To solve the problem, research in subsetting the language to allow more concise and efficient implementations has been conducted with promising preliminary results [Ueda and Morita 1990]. A separate effort of implementing KL1 by translating to C also indicated that reasonable performance can be obtained with very high portability [Chikayama 1992]. These results indicate the possibility of implementing the language on stock hardware efficiently for use in parallel software research. In addition to such an implementation, PIMOS, especially its software development environment, should also be ported to stock hardware to provide common basis of research and development of highly parallel knowledge information processing systems.

7 Conclusion

An overview of the research and development of the basic software for the parallel inference system of the FGCS project is given.

The system aims at establishing the basis of software technology for highly parallel computer systems. The research and development adopted a middle-out approach of designing a programming language first and then continuing the design both upwards to the application software and downwards to the hardware architecture simultaneously. The kernel language KL1 and the operating system PIMOS were designed and implemented.

The systems working on experimental parallel inference hardware Multi-PSI and a model of parallel inference machine PIM have been used in the research and development of application software since 1988. Our experiences have proved that the kernel language is expressive enough for describing an operating system for parallel processing systems and various application software. The features of the language that separated correctness and efficiency issues, along with the programming environment provided by the operating system, made empirical research of parallel software much easier than in conventional environments.

Further research in computation mapping is needed in future. Development of an efficient and comfortable environment on stock hardware is another important work to be done.

Acknowledgements

The design and implementation of KL1 and PIMOS for the parallel are collaborative work of many researchers too numerous to list here. The author would like to thank Kazunori Ueda for his helpful comments on an earlier version of this paper.

References

[Aikawa 1992 *et al.*] S. Aikawa, K. Mayumi, H. Kubo, F. Matsuzawa and T. Chikayama. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Archibald and Bare 1986] J. Archibald and J. L. Bare. Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In *ACM Trans. on Computer System*, Vol. 4, No. 4 (1986), pp. 273–298.

[Burton 1985] F. W. Burton. Speculative Computation, Parallelism and Functional Programming. In *IEEE Trans. Computers*, Vol. C-34, No. 12 (1985), pp. 1190–1193.

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 292–298.

[Chikayama 1991] T. Chikayama. For KL1 Programming without Tears. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 8–14. *in Japanese.*

[Chikayama 1992] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. To appear as an ICOT Tech. Report, ICOT, 1992.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 230–251.

[Clark and Gregory 1981] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 171–178.

[Clark and Gregory 1983] K. L. Clark and S. Gregory. PARLOG: A Parallel Logic Programming Language. Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, 1983.

[Clark and Gregory 1984] K. L. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 299–306.

[van Emden and de Lucena Filho 1982] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, K. L. Clark and S. -Å. Tärnlund (eds.), Academic Press, 1982, pp. 189–198.

[Foster 1987] I. Foster. Logic Operating Systems: Design Issues. In *Proc. Fourth Int. Conf. on Logic Programming*, J.-L. Lassez (ed.), MIT Press, Vol. 2, 1987, pp. 910–926.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki, N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1990, pp. 50–59.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 208–229.

[Hircsh *et al.* 1987] M. Hircsh, W. Silverman and Ehud Shapiro. Computation Control and Protection in the Logic System. In *Concurrent Prolog: Collected Papers*, Ehud Shapiro (ed.), MIT Press, Vol. 2, 1984, pp. 28–45.

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Inamura and Onishi 1990] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[Itoh *et al.* 1992] F. Itoh, T. Chikayama, T. Mori, M. Sato, T. Kato and T. Sato. The Design of the PIMOS File System. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Kondoh and Chikayama 1988] S. Kondoh and T. Chikayama. Macro Processing in Prolog. In *Proc. Fifth Int. Conf. and Symp. of Logic Programming*, 1988, pp. 466–480.

[Konishi *et al.* 1992] K. Konishi, T. Maruyama, A. Konagaya, K. Yoshida, T. Chikayama. Implementing Streams on Parallel Machines with Distributed Memory. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Levy and Silverschatz 1989] E. Levy and Z. Silberschatz. Distributed File Systems: Concepts and Examples. Tech. Report

TR-89-04, Dept. of Computer Science, The University of Texas at Austin, 1989.

[Maher 1987] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 858–876.

[Milner 1989] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Miyazaki *et al.* 1985] T. Miyazaki, A. Takeuchi and T. Chikayama. A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 110–118.

[Nakashima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine PSI-II. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987.

[Nitta *et al.* 1991] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Okumura and Matsumoto 1987] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 224–231.

[Shapiro 1983] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Tech. Report TR-003, ICOT, 1983.

[Shapiro 1986] E. Y. Shapiro. Systems Programming in Concurrent Prolog, In *Logic Programming and its Applications*, M. van Canegham and D. H. D. Warren (eds.), 1986, Ablex Publishing Co., 1986, pp. 50–74.

[Shapiro and Takeuchi 1983] E. Shapiro and A. Takeuchi. Object-oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol. 1, No. 1 (1983).

[Susaki and Chikayama 1991] K. Susaki and T. Chikayama. A Process-Oriented Language AYA upon KL1. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 117–125. *in Japanese.*

[Takeda *et al.* 1990] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 179–195.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Tamaki 1987] H. Tamaki. Stream-Based Compilation of Ground I/O Prolog into Committed-choice Languages. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 376–393.

[Ueda 1986] K. Ueda. Guarded Horn Clauses. In *Logic Programming '85*, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, 1986, pp. 168–179.

[Ueda 1987] K. Ueda. Making Exhaustive Search Programs Deterministic. In *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

[Ueda 1988a] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. In *Programming of Future Generation Computers*, M. Nivat. and K. Fuchi (eds.), North-Holland, 1988, pp. 441–456.

[Ueda 1988b] K. Ueda. Theory and Practice of Concurrent Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 165–166.

[Ueda 1990] K. Ueda. Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, 1990, pp. 87–94.

[Ueda and Chikayama 1985] K. Ueda and T. Chikayama. Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 119–126.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. In *The Computer Journal*, Vol. 33, No. 6 (1990) pp. 494–500.

[Ueda and Furukawa 1988] K. Ueda and K. Furukawa. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 582–591.

[Ueda and Morita 1990] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing.*

[Yashiro *et al.* 1992] H. Yashiro, T. Fujise, T. Chikayama, M. Matsuo, A. Hori and K. Wada. Resource Management Mechanism of PIMOS. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Yoshida and Chikayama 1990] K. Yoshida and T. Chikayama. A'UM: A Stream-Based Object-Oriented Language. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 127–157.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards an Integrated Knowledge-Base Management System

Overview of R&D on Databases and Knowledge-Bases in the FGCS Project

Kazumasa Yokota Hideki Yasukawa
Third Research Laboratory
Institute for New Generation Computer Technology (ICOT)
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
Tel: +81-3-3456-3069 Fax: +81-3-3456-1618
{kyokota,yasukawa}@icot.or.jp

Abstract

Knowledge representation languages and knowledge-bases play a key role in knowledge information processing systems. In order to support such systems, we have developed a knowledge representation language, *Quixote*, a database management system, *Kappa*, as the database engine, some applications on *Quixote* and *Kappa*, and two experimental systems for more flexible control mechanisms.

The whole system can be considered as under the framework of deductive object-oriented databases (DOODs) from a database point of view. On the other hand, from the viewpoint of the many similarities between database and natural language processing, it can also be considered to support situated inference in the sense of situation theory. Our applications have both of these features: molecular biological databases and a legal reasoning system, TRIAL, for DOOD and a temporal inference system for situated inference.

For efficient and flexible control mechanisms, we have developed two systems: cu-Prolog based on unfold/fold transformation of constraints and dynamical programming based on the dynamics of constraint networks.

In this paper, we give an overview of R&D activities for databases and knowledge-bases in the FGCS project, which are aimed towards an integrated knowledge-base management system.

1 Introduction

Since the Fifth Generation Computer System (FGCS) project started in 1982, many knowledge information processing systems have been designed and developed as part of the R&D activities in the framework of logic and parallelism. Such systems have various data and knowledge, that is, expected to be processed efficiently in the form of databases and knowledge-bases such as electronic dictionaries, mathematical databases, molecular biological databases, and legal precedent databases [1]. Representing and managing such large amounts of data and knowledge for these systems has been a major problem. Our activities on databases and knowledge-bases are also devoted to such data and knowledge under logic paradigm.

Since the late seventies, many data models have been proposed for extension of the relational model in order to overcome various disadvantages such as inefficient representation and inadequate query capability. Among their extensions, *deductive databases* attracted many researchers not only in logic communities but also in artificial intelligence communities, because of its logic platform and strong inference capability. Many efforts on deductive databases have defined the theoretical aspects of databases and have showed the powerful capability of query processing. However, from an application point of view, the data modeling capability is rather poor. This is mainly due to representation based on first-order predicates, which inherits the disadvantages of the relational model. On the other hand, *object-oriented databases* have become popular among extensions of the relational model for coping with 'new' applications such as CA*X* databases and multi-media databases. The flexibility and adaptability of object-orientation concepts should be examined also in the context of deductive databases, even if object-oriented databases have disadvantages such as poor formalism and semantic ambiguity.

[1] The boundary between *databases* and *knowledge-bases* is unclear and their usage depends on context. Most database communities prefer to use the term *database* even if databases store a set of rules and have an inference capability such as deduction and abduction: e.g., deductive databases, expert databases, and self-organizable databases. In this paper, we also use the term *database* according to this convention. The term *knowledge-base* in our title shows our view that an approach based on extensions of databases is a better way to *real* knowledge-bases than based on conventional *knowledge-bases* used by expert systems.

As it is appropriate for advanced applications to integrate their advantages, we proposed *deductive (and) object-oriented databases* (DOODs) [Yokota and Nishio 1989] [2] , where extensions of the relational model (or deductive databases and object-oriented databases) are considered from three directions: logic, data model, and computational model. The DOOD can be said to be a framework for such extensions. On the other hand, considering the many similarities between DOODs and natural language processing, the framework is also appropriate for situated inference in natural language processing. Such an observation leads us firmly towards an integrated knowledge-base management system over databases and knowledge representation languages.

In the FGCS project, we focus on DOODs as the target of knowledge-base management systems, based on the above observation, and have developed a knowledge-base (or knowledge representation) language *Quixote*, its database engine *Kappa*, and their applications. *Quixote* is a DOOD language. Also, a DOOD system based on *Quixote* has been implemented. We outline their features in Section 2. In order to process a large amount of data efficiently in the DOOD system, there should be a database engine at the lower layer. The engine is called *Kappa*, the data model of which is a nested relational model as a subclass of DOODs. For more efficient processing, a parallel database management system, *Kappa-P*, has been implemented on parallel inference machines. The data model and system are described in Section 3. We are also developing some applications on the DOOD system: a legal reasoning system (TRIAL), a molecular biological database, and a temporal inference system in natural language processing. An overview is given in Section 4. Together with the above works, we are engaged in R&D on more flexible control of logic programs: constraint transformation and dynamical programming, which are expected to be embedded in *Quixote*. We explain these in Section 5. Their relationship is shown in Figure 1. Finally we describe related works and future plans for further extensions of our knowledge-base management system.

2 Knowledge Representation Language (*Quixote*)

Our approach to knowledge-bases follows the previously mentioned deductive object-oriented databases (DOOD). The language, called *Quixote*, designed for the objective has various features [3] : a constraint logic programming language, a situated programming language, object-oriented database programming language, and a DOOD language, besides the features appearing in Figure 1.

2.1 Basic Concepts

Consider the example[Yoshida 1991] in Figure 2 for the genetic information processing system. In the figure,

```
object(ref('Patterson et al.(1981)'),
       1991/4/24,
       [kind(paper),
        authors(['D. Patterson',
                 'S. Graw',
                 'C. Jones'
                ]),
        title('Demonstration by somatic cell genetics of
               coordinate regulation of genes for two
               enzymes of puring synthesis assigned to
               human chromosome 21'),
        journal('Proc. Natl. Acad. Sci. USA'),
        volume(78),
        pages(405-409),
        year(1981)
       ]).
```

Figure 2: A Record (Term) in Prolog

the third argument of the term is peculiar: a tuple (in the form of a list) consisting of pairs of a label and its value. The *author* label has a set value, also, in the form of a list and some values might have a more complex value (another tuple). User programs must be responsible for such structure and unification among these terms. The reason why such a structure is necessary is that a record type (a scheme) cannot be decided in advance. That is, we can get only partial information for an object, because the object itself is not stable, generally. Such characteristics do not necessarily allow application of conventional normalization in the relational model to the design. By introducing an identity concept, such a record can be represented in the form of a set of binary relations, each of which has an identifier, however this is too inefficient in representation.

In *Quixote*, we introduce the concepts of an *object identifier* (*oid*) and a *property*, both of which are based on *complex object* constructors. The example in Figure 2 can be represented as in Figure 3 in *Quixote*. In the figure, the left hand side of "/" is an oid (called an *object term* (*o-term*) in *Quixote*) and the right hand side is the related properties. An *object* consists of an oid and its properties, and can be written as a set of *attribute terms* (*a-terms*) in *Quixote* with the same oid as follows:

$$o/[l_1 = a, l_2 = b] \iff o/[l_1 = a],\ o/[l_2 = b]$$

[2] International conferences were held in Kyoto and Munich [Kim *et al.* 1990, Delobel *et al.* 1991] to work towards such integration.

[3] See the details in [Yasukawa *et al.* 1992].

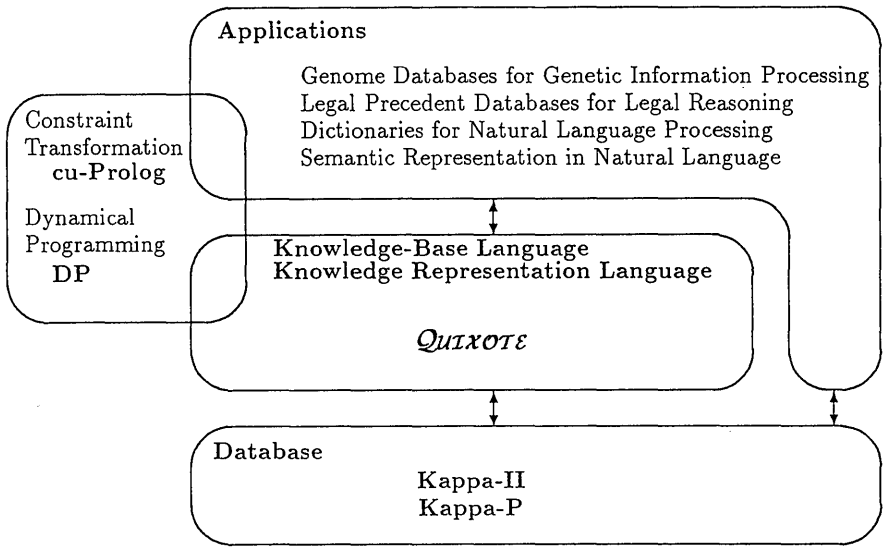


Figure 1: The Framework of a Knowledge-Base (DOOD) Management System of the FGCS Project

```
object[ref='Patterson et al (1981)']/
      [date=1991/4/24,
       kind=paper,
       authors={'D. Patterson',
                'S. Graw',
                'C. Jones'
                },
       title='Demonstration by somatic cell genetics of
              coordinate regulation of genes for two
              enzymes of puring synthesis assigned to
              human chromosome 21',
       journal='Proc. Natl. Acad. Sci. USA',
       volume=78,
       pages=405-409,
       year=1981
      ].
```

Figure 3: An Object in QUIXOTE

Such description is effective for processing partial information. Attributes in an o-term are intrinsic for the object:

$$o[l=c]/[l_1=a, l_2=b] \iff o[l=c]/[l=c, l_1=a, l_2=b]$$

where the right hand side, "$[\cdots]$", of "/" is called the *attribution* of $o[l=c]$. An attribute in an o-term is called an *intrinsic* (or immutable) attribute and an attribute appearing only in the attribution is called an *extrinsic* (or mutable) attribute [4].

[4]We sometimes abuse the terms *attribute* and *property*. Although both an *attribute* and a *property* are, usually, a pair of a label and a (possibly complex) value, an *attribute* is frequently used in the context of record structure, while a *property* is frequently used in the context of object structure. In QUIXOTE, a pair of a label and a value (or a triple of a label, an operator, and a value) is called an *attribute*, however, in the context of inheritance, we use *property* inheritance as a convention. As only extrinsic attributes are inherited in QUIXOTE, as mentioned later, extrinsic attributes are simply called properties. Furthermore, there is a case where an attribute means only a label, as in an attribute-value pair, the meaning, however, is usually clear in the context.

Another problem is the expressive power of oids and properties. First, along the style of logic programming, an oid can be defined intensionally by a set of rules as follows:

$$\begin{aligned} path[from=X, to=Y] &\Leftarrow arc[from=X, to=Y]. \\ path[from=X, to=Y] &\Leftarrow arc[from=X, to=Z], \\ &\qquad path[from=Z, to=Y]. \end{aligned}$$

In this program, $path[from=X, to=Y]$ is transitively defined from a set of facts such as $arc[from=a, to=b]$ and so on, and the oid is generated by instantiating X and Y as the result of execution of the program. This guarantees that an object can have a unique oid even if the object is generated in different environments. Furthermore, in order to define a circular path, we must introduce a *tag* and represent a, so-called, complex object with a set and a tuple constructor.

$$\begin{aligned} X@o[l=X] &\iff X|\{X=o[l=X]\} \\ o[l=\{a, \cdots, b\}] &\iff o[l=a] \wedge \cdots \wedge o[l=b] \end{aligned}$$

The first example shows that a variable X is an oid with a constraint, $X=o[l=X]$. The second shows that a set in an o-term can be decomposed into conjunction of o-terms without a set constructor.

On the other hand, properties might be indefinite, that is, only in the form of constraints. We introduce the following operators between a label and a (set of) value, and transform them into a set of constraints by introducing dot notation:

$$\begin{aligned} o/[l=a] &\iff o/|\{o.l \cong a\} \\ o/[l\rightarrow a] &\iff o/|\{o.l \sqsubseteq a\} \\ o/[l\leftarrow a] &\iff o/|\{o.l \sqsupseteq a\} \\ o/[l=\{a,\cdots,b\}] &\iff o/|\{o.l \cong_H \{a,\cdots,b\}\} \\ o/[l\rightarrow \{a,\cdots,b\}] &\iff o/|\{o.l \sqsubseteq_H \{a,\cdots,b\}\} \\ o/[l\leftarrow \{a,\cdots,b\}] &\iff o/|\{o.l \sqsupseteq_H \{a,\cdots,b\}\} \end{aligned}$$

The right hand side of "/|" is a set of constraints about properties, where $\sqsubseteq_H$ and $\sqsupseteq_H$ are a partial order generated by Hoare ordering defined by $\sqsubseteq$ and $\sqsupseteq$, respectively, and $\cong_H$ is the equivalence relation. If an attribute of a label l is not specified for an object o, o is considered to have a property of l without any constraint.

The semantics of oids is defined on a set of labeled graphs as a subclass of hypersets[Aczel 1988]: an oid is mapped to a labeled graph and an attribute is related to a function on a set of labeled graphs. In this sense, attributes can be considered *methods* to an object as in F-logic[Kifer and Lausen 1989].

The reason for adopting a hyperset theory as the semantic domain is to handle an infinite data structure. The details can be found in [Yasukawa *et al.* 1992].

2.2 Subsumption Relation and Property Inheritance

Given a partial order relation in a set of basic (non-structural) objects, we can constitute a lattice in a set of ground object terms, the order of which is called the *subsumption relation* $\sqsubseteq$. This is already used as a relation for properties as constraints. According to the relation, properties are inherited downward and/or upward among objects. A general *property inheritance* rule is as follows:

$$o_1 \sqsubseteq o_2 \ \supset\ o_1.l \sqsubseteq o_2.l$$

where intrinsic attributes are out of inheritance. According to the rule, we can get the following:

$$\begin{aligned} o_1 \sqsubseteq o_2,\ o_2/|\{o_2.l \sqsubseteq a\} &\implies o_1/|\{o_1.l \sqsubseteq a\} \\ o_1 \sqsubseteq o_2,\ o_1/|\{o_1.l \sqsupseteq a\} &\implies o_2/|\{o_2.l \sqsupseteq a\} \\ o_1 \sqsubseteq o_2 \sqsubseteq o_3,\ o_2/|\{o_2.l \cong a\} &\implies o_1/|\{o_1.l \sqsubseteq a\}, \\ & \qquad\quad o_3/|\{o_3.l \sqsupseteq a\} \end{aligned}$$

where it can be noted that $o_2/|\{o_2.l \sqsubseteq a\}$ is $o_2/[l\rightarrow a]$: that is, property inheritance is constraint inheritance. In complex o-terms, intrinsic attributes cause the exception of property inheritance:

$$o[l=a] \sqsubseteq o,\ o/[l\rightarrow b] \implies o[l=a]/[l=a]$$

Multiple inheritance is defined upward and downward as the merging of constraints:

$$\begin{aligned} o_1 \sqsubseteq o_2,\ o_1 \sqsubseteq o_3,\ o_2/[l\rightarrow a],\ o_3/[l\rightarrow b] & \\ \implies o/[l\rightarrow \mathrm{meet}(a,b)] & \\ o_1 \sqsupseteq o_2,\ o_1 \sqsupseteq o_3,\ o_2/[l\leftarrow a],\ o_3/[l\leftarrow b] & \\ \implies o/[l\leftarrow \mathrm{join}(a,b)] & \end{aligned}$$

where a set of constraints are reduced by the constraint solver.

2.3 Program and Database

A *module* concept is introduced in order to classify knowledge and handle (local) inconsistencies. Let m be a *module identifier* (*mid*) (syntactically the same as an o-term) and a be an a-term, then $m{:}a$ is a *proposition*, which means that m *supports* a. Given a mid m, an a-term a, and propositions $p_1,\cdots,p_n$, a *rule* is defined as follows:

$$m :: a \Leftarrow p_1,\cdots,p_n.$$

which means that a module with a mid m has a rule such that if $p_1,\cdots,p_n$ hold, a holds in a module with a mid m. If a mid is omitted in p_i, m is taken as the default and if m is omitted, the rule is held in all modules. a is called a *head* and $p_1,\cdots,p_n$ is called a *body*. As an a-term can be separated into an o-term and a set of constraints, the rule can be rewritten as follows:

$$m :: o/|C_H \Leftarrow m_1{:}o_1,\cdots,m_n{:}o_n||C_B.$$

where $a \cong o/|C_H$, $p_i \cong m_i : o_i|C_i$, and $C_B = C_1 \cup \cdots \cup C_n$. C_H is a *head constraint* and C_B is a *body constraint*. Their domain is a set of labeled graphs. Note that constraints by a-terms in a body can be included in C_B. Compared with conventional constraint logic programming, a head constraint is new.

A *module* is defined as a set of rules with the same mid. We define the acyclic relation among modules, a *submodule* relation. This works for *rule inheritance* as follows:

$$\begin{aligned} m_1 &\sqsupseteq_S m_2 \\ m_3 &\sqsupseteq_S m_4 \cup (m_5 \setminus m_6) \end{aligned}$$

where m_1 inherits a set of rules in m_2, and m_3 inherits a set of rules defined by set operations such as $m_4 \cup (m_5 \setminus m_6)$. Set operations such as intersection and difference are syntactically evaluated. Even if a module is parametric, that is, the mid is an o-term with variables, the submodule relation can be defined. In order to treat the exception of rule inheritance, each rule has properties such as *local* and *overriding*: a local rule is not inherited to other modules and an overriding rule obstructs the inheritance of rules with the same head from other modules.

A *program* or a *database* is defined as a set of rules with definitions of subsumption and submodule relations. Clearly, a program can be also considered as a set of modules, where an object may have different properties if it exists in different modules. Therefore, we can classify a knowledge-base into different modules and define a submodule relation among them. If a submodule relation is not defined among two modules, even transitively, an object with the same oid may have different (or even inconsistent) properties in its modules. The semantics of a program is defined on the domain of pairs of labeled graphs corresponding to a mid and an o-term. In this framework, we can classify a large-scaled knowledge-base, which might have inconsistencies, and store it in a *QUIXOTE* database.

2.4 Updating and Persistence

QUIXOTE has a concept of nested transaction and allows two kinds of database update:

1) *incremental insert* of a *database* when issuing a query, and

2) *dynamic insert* and *delete* of *o-terms* and *a-terms* during query processing.

We can issue a query with a new database to be added to the existing database. 1) corresponds to the case. For example, consider the following sequence of queries to a database DB:

query sequence to DB		equivalent query
?- `begin_transaction`.		
?- Q_1 `with` DB_1.	$\Longleftrightarrow$	?- Q_1 `to` $DB \cup DB_1$
?- `begin_transaction`.		
?- Q_2 `with` DB_2.	$\Longleftrightarrow$	?- Q_2 `to` $DB \cup DB_1 \cup DB_2$
?- `abort_transaction`.		
?- Q_3 `with` DB_3.	$\Longleftrightarrow$	?- Q_1 `to` $DB \cup DB_1 \cup DB_3$
?- Q_4.	$\Longleftrightarrow$	?- Q_1 `to` $DB \cup DB_1 \cup DB_3$
?- `end_transaction`.		

After successful execution of the above sequence, DB is changed to $DB \cup DB_1 \cup DB_3$. Each DB_i may have definitions of a subsumption relation or a submodule relation, which are merged into definitions of the existing database, If necessary, the subsumption or submodule hierarchy is reconstructed. By rolling back the transaction, such a mechanism can also be used as hypothesis reasoning.

2) makes it possible to update an o-term or its (mutable) properties during query processing, where transactions are located as subtransactions of a transaction in 1). In order to guarantee the semantics of update, so-called AND- and OR-parallel executions are inhibited. For example, the following is a simple rule for updating an employees' salary:

$$
\begin{array}{l}
pay[year=1992, dept=X]/[raise=Y] \\
\quad \Leftarrow \texttt{begin\_transaction}; \\
\quad\quad employee[num=Z]/[dept=X, salary=W]; \\
\quad\quad -employee[num=Z]/[salary=W]; \\
\quad\quad +employee[num=Z]/[salary=New]; \\
\quad\quad \texttt{end\_transaction} \\
\quad\quad ||\{New = W * Y\}.
\end{array}
$$

where ";" specifies sequential execution in order to suppress AND-parallel execution, "+" means insert, and "−" means delete.

Except for the objects to be deleted or rolled back during query processing, all (extensional or intensional) objects in a *QUIXOTE* program are guaranteed to be persistent. Such persistent objects are stored in the underlying database management system (explained in the next section) or a file system.

2.5 Query Processing and the System

QUIXOTE is basically a constraint logic programming language with object-orientation features such as object identity, complex object, encapsulation, type hierarchy, and methods. However, this query processing is different from conventional query processing because of the existence of oids and head constraints. For example, consider the following program:

$$
\begin{array}{l}
lot[num=X]/[prize_1 \rightarrow a] \Leftarrow X \sqsubseteq 2n. \\
lot[num=X]/[prize_2 \rightarrow b] \Leftarrow X \sqsubseteq 3n. \\
lot[num=X]/[prize_1 \rightarrow c] \Leftarrow X \sqsubseteq 5n.
\end{array}
$$

where $2n$ is a type with a multiple of two. Given a query $?\text{-}lot[num = 30]/[prize_1 = X, prize_2 = Y]$, the answer is $X \sqsubseteq \text{meet}(a, c)$ and $Y \rightarrow b$, that is,

$$lot[num=30]/[prize_1 \rightarrow \text{meet}(a, c), prize_2 \rightarrow b].$$

First, because of the existence of oids, all rules which possibly have the same oid must be evaluated and merged if necessary. Therefore, in *QUIXOTE*, a query is always processed in order to obtain all solutions. Secondly, as a rule in *QUIXOTE* has two kinds of constraints, a head constraint and a body constraint, each of which consists of equations and inequations of dotted terms besides the variable environment, the derivation process is different from conventional constraint logic programming:

$$(G_0, \emptyset) \rightarrow \cdots \rightarrow (G_i, C_i) \rightarrow \cdots \rightarrow (\emptyset, C_n)$$

where G_i is a set of subgoals and C_i is a set of constraints of the related variables. On the other hand, in *QUIXOTE*, each node in the derivation sequence is (G, A, C), where G is a set of *subgoals*, A is a set of *assumptions* consisting of a body constraint of dotted terms, and C is a set of *conclusions* as a set of constraints consisting of a head constraint and a variable environment. Precisely speaking, the derivation is not a sequence but a directed acyclic graph in

Quixote, because some subsumption relation among assumptions and constraints might force the two sequences to merge: for example, (G, A, C) and (G, A, C') are merged into $(G, A, C \cup C')$. Therefore, the derivation is shown in Figure 4, where the environment to make

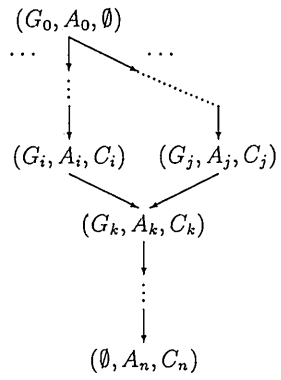


Figure 4: Derivation in *Quixote*

it possible to merge two sequences is restricted: only results by the, so-called, OR-parallel that includes rules inherited by subsumption relation among rule heads can be merged innermostly. The current implementation of query processing in *Quixote* is based on a tabular method such as OLDT in order to obtain all solutions. Sideways information passing is also implemented by considering not only binding information but also property inheritance.

We list some features of the *Quixote* system:

- A *Quixote* program is stored in persistent storage in the form of both the 'source' code and the 'object' code, each of which consists of four parts: control information, subsumption relation, submodule relation, and a set of rules. Persistence is controlled by the *persistence manager*, which switches where programs should be stored. A set of rules in the 'object' code is optimized to separate extensional and intensional databases as in conventional deductive databases.

- When a user builds a huge database in *Quixote*, it can be written as a set of small databases independently of a module concept. These can be gathered into one database, that is, a database can be reused in another database.

- When a user utilizes data and knowledge in *Quixote*, multiple databases can be accessed simultaneously through the *Quixote server*, although the concurrency control of the current version of *Quixote* is simply implemented.

- Users can use databases through their application programs in ESP [Chikayama 1984] or KL1 [Ueda and Chikayama 1990], and through the specific window interface called *Qmacs*.

The environment is shown in Figure 5.

The first version of *Quixote* was released in December, 1991. A second version was released in April, 1992. Both versions are written in KL1 and work on parallel inference machines (PIMs) [Goto *et al.* 1988] and its operating system (PIMOS) [Chikayama *et al.* 1988].

3 Advanced Database Management System (Kappa)

In order to process a large database in *Quixote* efficiently, a database engine called *Kappa* has been developed [5]. In this section, we explain its features.

3.1 Nested Relation and *Quixote*

The problem is which part of *Quixote* should be supported by a database engine because enriched representation is a trade-off in efficient processing. We intend for the database engine to be able to, also, play the role of a practical database management system. Considering the various data and knowledge in our knowledge information processing environment, we adopt an *extended nested relational model*, which corresponds to the class of an o-term without infinite structure in *Quixote*. The term "*extended*" means that it supports a new data type such as Prolog term and provided extensibility as the system architecture for various applications. The reason why we adopt a nested relational model is, not surprisingly, to achieve efficient representation and efficient processing.

Intuitively, a *nested relation* is defined as a subset of a Cartesian product of domains or other nested relations:

$$\begin{aligned} NR &\subseteq E_1 \times \cdots \times E_n \\ E_i &::= D \mid 2^{NR} \end{aligned}$$

where D is a set of atomic values. That is, the relation may have a hierarchical structure and a set of other relations as a value. This corresponds to the introduction of tuple and set constructors. From the viewpoint of syntactical and semantical restrictions, there are various subclasses. Extended relational algebra are defined to each of these.

In Kappa's nested relation, a set constructor is used only as an abbreviation of a set of normal relations as follows:

$$\begin{aligned} &\{r[l_1 = a, l_2 = \{b_1, \cdots, b_n\}]\} \\ &\quad \Longleftrightarrow \quad \{r[l_1 = a, l_2 = b_1], \cdots, r[l_1 = a, l_2 = b_n]\} \end{aligned}$$

[5] See the details in [Kawamura *et al.* 1992].

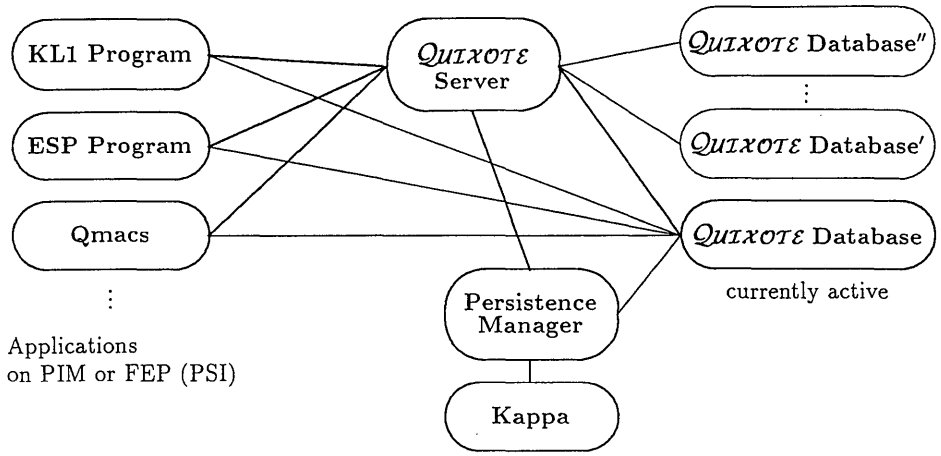


Figure 5: Environment of *QUIXOTE*

The operation of "⇒" corresponds to an *unnest* operation, while the opposite operation ("⇐") corresponds to a *nest* or *group-by* operation, although "⇐" is not necessarily congruent for application of nest or group-by operation sequences, That is, in Kappa, the semantics of a nested relation is the same as the coresponding relation without set constructors. The reason for taking such semantics is to retain first order semantics for efficient processing and to remain compatible to widely used relational databases. Given a nested tuple nt, let the corresponding set of tuples without a set constructor be $\underline{nt}$. Let a nested relation be

$$NR = \{nt_1, \cdots, nt_n\}$$
$$\text{where } \underline{nt_i} = \{t_{i1}, \cdots, t_{ik}\} \text{ for } i = 1, \cdots, n,$$

then the semantics of NR is

$$\bigcup_{i=1}^{n} \underline{nt_i} = \{t_{11}, \cdots, t_{1k}, \cdots, t_{n1}, \cdots, t_{nk}\}.$$

Extended relational algebra to this nested relational database is defined in Kappa and produces results according to the above semantics, which guarantees to produce the same result to the corresponding relational database, except for treatment of the label hierarchy.

A query can be formulated as a first order language, we, generally, consider this in the form of a rule constructed by nested tuples. As the relation among facts in a database is conjunctive from a proof-theoretic point of view, the semantics of a rule is clear according to the above semantics. For example, the following rule

$$r[l_1 = X, l_2 = \{a, b, c\}]$$
$$\Leftarrow B, r'[l_2 = Y, l_3 = \{d, e\}, l_3 = Z], B'.$$

can be transformed into the following set of rules without set constructors:

$$r[l_1 = X, l_2 = a]$$
$$\Leftarrow B, r'[l_2 = Y, l_3 = d, l_3 = Z], r'[l_2 = Y, l_3 = e, l_3 = Z], B'.$$
$$r[l_1 = X, l_2 = b]$$
$$\Leftarrow B, r'[l_2 = Y, l_3 = d, l_3 = Z], r'[l_2 = Y, l_3 = e, l_3 = Z], B'.$$
$$r[l_1 = X, l_2 = c]$$
$$\Leftarrow B, r'[l_2 = Y, l_3 = d, l_3 = Z], r'[l_2 = Y, l_3 = e, l_3 = Z], B'.$$

That is, each rule can also be unnested. The point of efficiently processing Kappa relations is to reduce the number of unnest and nest operations: that is, to process sets as directly as possible.

Under the semantics, query processing to nested relations is different from conventional procedures in logic programming. For example, consider a simple database consisting of only one tuple:

$$r[l_1 = \{a, b\}, l_2 = \{b, c\}].$$

For a query ?-$r[l_1 = X, l_2 = X]$, we can get $X = \{b\}$, that is, an intersection of $\{a, b\}$ and $\{b, c\}$. That is, a concept of unification should be extended. In order to generalize such a procedure, we must introduce two concepts into the procedural semantics[Yokota 1988]:

1) *Residue Goals*
Consider the following program and a query:

$$r[l = S'] \Leftarrow B.$$
$$?\text{-}r[l = S].$$

If $S \cap S'$ is not an empty set during unification between $r[l = S]$ and $r[l = S']$, new subgoals are to be $r[l = S \setminus S'], B$. That is, a residue subgoal $r[l = S \setminus S']$ is generated if $S_1 \setminus S_2$ is not an empty set, otherwise the unification fails. Note that there might be residue subgoals if there are multiple set values.

2) *Binding as Constraint*
Consider the following database and a query:

$$r_1[l_1 = S_1].$$

$$r_2[l_2 = S_2].$$
$$?\text{-}r_1[l_1 = X], r_2[l_2 = X].$$

Although we can get $X = S_1$ by unification between $r_1[l_1 = X]$ and $r_1[l_1 = S_1]$ and a new subgoal $r_2[l_2 = S_1]$, the subsequent unification results in $r_2[l_2 = S_1 \cap S_2]$ and a residue subgoal $r_2[l_2 = S_1 \setminus S_2]$. Such a procedure is wrong, because we should have an answer $X = S_1 \cap S_2$. In order to avoid this situation, the binding information is temporary and plays the role of constraints to be retained:

$$r_1[l_1 = X], r_2[l_2 = X]$$
$$\Longrightarrow r_2[l_2 = X] || \{X \subset S_1\}$$
$$\Longrightarrow || \{X \subset S_1 \cap S_2\}.$$

There remains one problem where the unique representation of a nested relation is not necessarily decided in the Kappa model, as already mentioned. In order to decide a unique representation, each nested relation has a sequence of labels to be nested in Kappa.

As the procedural semantics of extended relational algebra in Kappa is defined by the above concepts, a Kappa database does not necessarily have to be *normalized* also in the sense of nested relational models, in principle. That is, it is unnecessary for users to be conscious of the row nest structure.

Furthermore, nested relational model is well known to reduce the number of relations in the case of multivalue dependency. Therefore, the Kappa model guarantees more efficient processing by reducing the number of tuples and relations, and more efficient representation by complex construction than the relational model.

3.2 Features of Kappa System

The nested relational model in Kappa has been implemented. This consists of a sequential database management system *Kappa-II* [Yokota *et al.* 1988] and a parallel database management system *Kappa-P* [Kawamura *et al.* 1992]. Kappa-II, written in ESP, works on sequential inference machines (PSIs) and its operating system (SIMPOS). Kappa-P, written in KL1, works on parallel inference machines (PIMs) and its operating system (PIMOS). Although their architectures are not necessarily the same because of environmental differences, we explain their common features in this subsection,

- *Data Type*
  As Kappa aims at a database management system (DBMS) in a knowledge information processing environment, a new data type, *term*, is added. This is because various data and knowledge are frequently represented in the form of terms. Unification and matching are added for their operations. Although unification-based relational algebra can emulate the derivation in logic programming, the features are not supported in Kappa because the algebra is not so efficient. Furthermore, Kappa discriminates one-byte character (ASCII) data from two-byte character (JIS) data as data types. It contributes to the compression of huge amounts of data such as genetic sequence data.

- *Command Interfaces*
  Kappa provides two kinds of command interface: *basic commands* as the low level interface and extended relational algebra as the high level interface. In many applications, the level of extended relational algebra, which is expensive, is not always necessary. In such applications, users can reduce the processing cost by using basic commands.

  In order to reduce the communication cost between a DBMS and a user program, Kappa provides user-definable commands, which can be executed in the same process of the Kappa kernel (in Kappa-II) or the same node of each local DBMS (in Kappa-P, to be described in the next subsection).

  The user-definable command facility helps users design any command interface appropriate for their application and makes their programs run efficiently. Kappa's extended relational algebra is implemented as parts of such commands although it is a built-in interface.

- *Practical Use*
  As already mentioned, Kappa aims, not only at a database engine of *Quixote*, but also at a practical DBMS, which works independently of *Quixote*. To achieve this objective, there are several extensions and facilities. First, new data types, besides the data types mentioned above, are introduced in order to store the environment under which applications work. There are *list*, *bag*, and *pool*. They are not, however, supported fully in extended relational algebra because of semantic difficulties.

  Kappa supports the same interface to such data types as in SIMPOS or PIMOS.

  In order to use Kappa databases from windows, Kappa provides a user-friendly interface, like a spreadsheet, which provides an ad hoc query facility including update, a browsing facility with various output formats and a customizing facility.

- *Main Memory Database*
  Frequently accessed data can be loaded and re-

tained in the main memory as a main memory database. As such a main memory database was designed only for efficient processing of temporary relations without additional burdens in Kappa, the current implementation does not support conventional mechanisms such as deferred update and synchronization. In Kappa-P, data in a main memory database are processed at least three times more efficiently than in a secondary storage database.

From an implementational point of view, there are several points for efficient processing in Kappa. We explain two of them:

- *ID Structure and Set Operation*
  Each nested tuple has a unique tuple identifier (*ntid*) in a relation, which is treated as an 'object' to be operated explicitly. Abstractly speaking, there are four kinds of 'object's, such as a *nested tuple*, an *ntid*, a *set* whose element is a ntid, and a *relation* whose element is a nested tuple. Their commands for transformation are basically supported, as in Figure 6, although the set

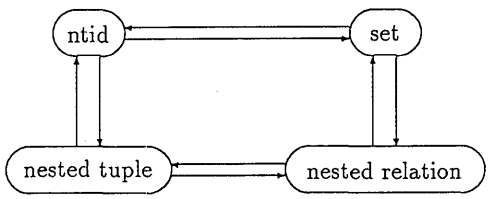


Figure 6: 'Object's in Kappa and Basic Operations

  is treated as a *stream* in Kappa-P. Most operations are processed in the form of an ntid or a set.

  In order to process a selection result, each subtuple in a nested tuple also has a *sub-ntid* virtually. Set operations (including unnest and nest operation) are processed mainly in the form of a (sub-)ntid or a set without reading the corresponding tuples.

- *Storage Structure*
  A nested tuple, which consists of unnested tuples in the semantics, is also considered as a set of unnested tuples to be accessed together. So, a nested tuple is compressed *without decomposition* and stored on the same page, in principle, in the secondary storage. For a huge tuple, such as a genetic sequence, contiguous pages are used. In order to access a tuple efficiently, there are two considerations: how to locate the necessary tuple efficiently, and how to extract the necessary attributes efficiently from the tuple. As in Figure 7, Kappa is equipped with an efficient address translation table between an ntid and a logical page (*lp*), and between a logical page and a physical page (*pp*). This table is used by the underlying file system. For extraction purposes, each node of

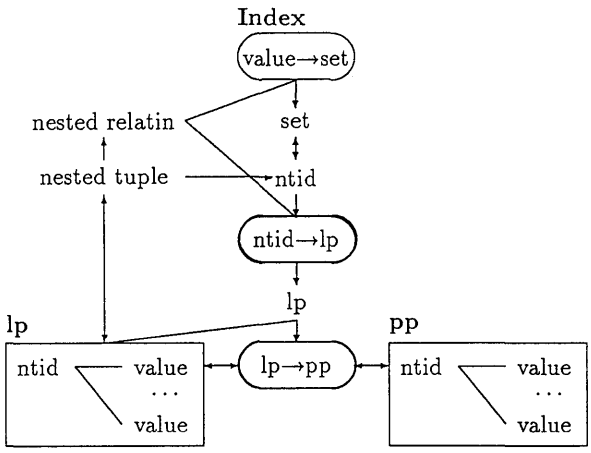


Figure 7: Access Network for Secondary DBMS

  a nested tuple has a local pointer and counters in the compressed tuple, although there is a trade-off in update operations' efficiency.

  Each entry in an index reflects the nested structure: that is, it contains any necessary sub-ntids. The value in the entry can be the result of string operations such as substring and concatenation of the original values, or a result extracted by a user's program.

3.3 Parallel Database Management System (Kappa-P)

Kappa-P has various unique features as a parallel DBMS. In this subsection, we give a brief overview of them.

The overall configuration of Kappa-P is shown in Figure 8. There are three components: an *interface (I/F) process*, a *server DBMS*, and a *local DBMS*. An I/F process, dynamically created by a user program, mediates between a user program and (server or local) DBMSs by *streams*. A server DBMS has a global map of the location of local DBMSs and makes a user's stream connect directly to an appropriate local DBMS (or multiple local DBMSs). In order to avoid a bottleneck in communication, there might be many server DBMSs with replicates global maps. A local DBMS can be considered as a single nested relational DBMS, corresponding to Kappa-II, where users' data is stored.

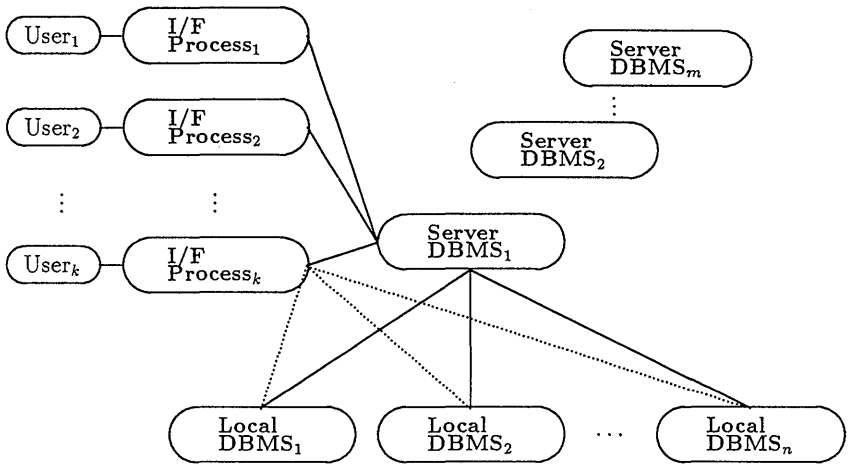


Figure 8: Configuration of Kappa-P

Users' data may be distributed (even horizontally partitioned) or replicated into multiple local DBMSs. If each local DBMS is put in a shared memory parallel processor, called a *cluster* in PIM, each local DBMS works in parallel. Multiple local DBMSs are located in each node of distributed memory parallel machine, and, together, behave like a distributed DBMS.

User's procedures using extended relational algebra are transformed into procedures written in an intermediate language, the syntax of which is similar to KL1, by an interface process. During the transformation, the interface process decides which local DBMS should be the coordinator for the processing, if necessary. Each procedure is sent to the corresponding local DBMS, and processed there. Results are gathered in the coordinator and then processed.

Kappa-P is different from most parallel DBMS, in that most users' applications also work in the same parallel inference machine. If Kappa-P coordinates a result from results obtained from local DBMSs, as in conventional distributed DBMSs, even when such coordination is unnecessary, the advantages of parallel processing are reduced. In order to avoid such a situation, the related processes in a user's application can be dispatched to the same node as the related local DBMS as in Figure 9. This function contributes not only to efficient processing but also to customization of the command interface besides the user-defined command facility.

4 Applications

We are developing three applications on *Quixote* and Kappa, and give an overview of each research topic in this section.

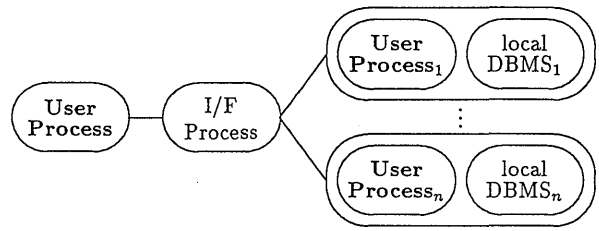


Figure 9: User's Process in Kappa Node

4.1 Molecular Biological Database

Genetic information processing systems are very important not only from scientific and engineering points of view but also from a social point of view, as shown in the Human Genome Project. Also, at ICOT, we are engaged in such systems from thr viewpoint of knowledge information processing. In this subsection, we explain such activities, mainly focusing on molecular biological databases in *Quixote* and Kappa [6].

4.1.1 Requirements for Molecular Biological Databases

Although the main objective of genetic information processing is to design proteins as the target and to produce them, there remain too many technical difficulties presently. Considering the whole of proteins, we are only just able to gather data and knowledge with much noise.

In such data and knowledge there are varieties such as sequences, structures, and functions of genes and proteins, which are mutually related. A gene in the

[6] See the details in [Tanaka 1992].

genetic sequence (DNA) in the form of a *double helix* is copied to a mRNA and translated into an amino acid sequence, which becomes a part (or a whole) of a protein. Such processes are called the Central Dogma in biology. There might be different amino acids even with the same functions of a protein. The size of a unit of genetic sequence data ranges from a few characters to around 200 thousand, and will become longer as genome data is gradually analyzed fyrther. The size of a human genome sequence equals about 3 billion characters. As there are too many unknown proteins, the sequence data is fundamental for homology searching by a pattern called a motif and for multiple alignment among sequences for prediction of the functions of unknown proteins from known ones.

There are some problems to be considered for molecular biological databases:

- how to store large values, such as sequences, and process them efficiently,
- how to represent structure data and what operations to apply them,
- how to represent functions of protein such as chemical reactions, and
- how to represent their relations and link them.

From a database point of view, we should consider some points in regard to the above data and knowledge:

- representation of complex data as in Figure 2,
- treatment of partial or noisy information in unstable data,
- inference rules representing functions, as in the above third item, and inference mechanisms, and
- representation of hierarchies such as biological concepts and molecular evolution.

After considering the above problems, we choose to build such databases on a DOOD (*Quixote*, conceptually), while a large amount of simple data is stored in Kappa-P and directly operated through an optimized window interface, for efficient processing. As cooperation with biologists is indispensable in this area, we also implemented an environment to support them. The overall configuration of the current implementation is shown in Figure 10.

4.1.2 Molecular Biological Information in *Quixote* and Kappa

Here, we consider two kinds of data as examples: sequence data and protein function data.

First, consider a DNA sequence. Such data does not need inference rules, but needs a strong capability for homology searching. In our system, such data is stored

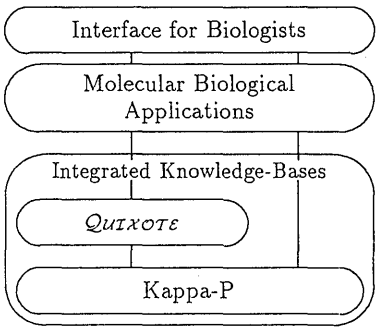


Figure 10: Integrated System on *Quixote* and Kappa

directly in Kappa, which supports the storage of much data as is and creates indexes from the substrings extracted from the original by a user program. Sequence-oriented commands for information retrieval, which use such indexes, can be embedded into Kappa as user-defined commands. Furthermore, since the complex record shown in Figure 3 is treated like a nested relation, the representation is also efficient. Kappa shows its effectiveness as a practical DBMS.

Secondly, consider a chemical reaction of enzymes and co-enzymes, whose scheme is as follows:

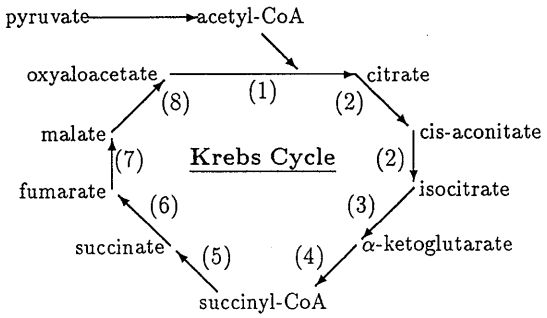
$$\text{Sources} + \text{Co-enzymes} \overset{\text{Enzymes}}{\underset{\text{Environments}}{\Longrightarrow}} \text{Products}$$

As an example of metabolic reaction, consider the Krebs cycle in Figure 11. Chemical reactions in the Krebs cycle are written as a set of facts in *Quixote* as in Figure 12. In the figure, $o_1 \sqsubseteq o_2/[\cdots]$ means $o_1/[\cdots]$ and $o_1 \sqsubseteq o_2$. In order to obtain a reaction chain (path) from the above facts, we can write the following rules in *Quixote*:

$$\begin{aligned} &reaction[from{=}X, to{=}Y] \\ &\quad \Leftarrow W \sqsubseteq reaction/[sources^{+} \leftarrow X, \\ &\qquad\qquad products^{+} \leftarrow Z], \\ &\qquad reaction[from{=}X, to{=}Y] \\ &\qquad ||\{\{X, Y, Z\} \sqsubseteq reaction\}. \\ &reaction[from{=}X, to{=}X] \\ &\quad \Leftarrow ||\{X \sqsubseteq reaction\}. \end{aligned}$$

Although there are a lot of difficulties in representing such functions, *Quixote* makes it possible to write them down easily.

Another problem is how to integrate a Kappa database with a *Quixote* database. Although one of the easiest ways is to embed the Kappa interface into *Quixote*, it costs more and might destroy a uniform representation in *Quixote*. A better way would be to manage common oids both in Kappa and in *Quixote*, and guarantee the common object, however we have



ENZYMES
(1) citrate synthase
(2) aconitate
(3) isocitrate dehydrogenase
(4) α-ketoglutarate dehydrogenase complex
(5) succinyl-CoA synthetase
(6) succinate dehydrogenase
(7) fumarase
(8) malate dehydrogenas

Figure 11: Krebs Cycle in Metabolic Reaction

not implemented such a facility in Kappa. The current implementation puts the burden of the uniformity on the user, as in Figure 10.

4.2 Legal Reasoning System (TRIAL)

Recently, legal reasoning has attracted much attention from researchers in artificial intelligence, with high expectations for its *big* application. Some prototype systems have been developed. We also developed such a system as one of the applications of our DOOD system [7].

4.2.1 Requirements for Legal Reasoning Systems and TRIAL

First, we explain the features of legal reasoning. The analytical legal reasoning process is considered as consisting of three steps: *fact findings*, *statutory interpretation*, and *statutory application*.

Although fact findings is very important as the starting point, it is too difficult for current technologies. So, we assume that new cases are already represented in the appropriate form for our system. Statutory interpretation is one of the most interesting themes from an artificial intelligence point of view. Our legal reasoning system, TRIAL, focuses on statutory interpretation as well as statutory application.

[7]See the details in [Yamamoto 1990], although the new version is revised as in this section.

$$
\begin{array}{l}
krebs\_cycle :: \{\{ \\
\quad krebs1 \sqsubseteq reaction/ \\
\qquad [sources^{+} \leftarrow \{acetylcoa, oxaloacetate\}, \\
\qquad\ products^{+} \leftarrow \{citrate, coa\}, \\
\qquad\ enzymes \leftarrow citrate\_synthase, \\
\qquad\ energy = -7.7]. \\
\quad krebs2 \sqsubseteq reaction/ \\
\qquad [sources^{+} \leftarrow citrate, \\
\qquad\ products^{+} \leftarrow \{isocitrate, h2o\}, \\
\qquad\ enzymes \leftarrow aconitase]. \\
\quad \vdots \\
\quad krebs8 \sqsubseteq reaction/ \\
\qquad [sources^{+} \leftarrow malate, \\
\qquad\ products^{+} \leftarrow oxaloacetate, \\
\qquad\ enzymes \leftarrow malate\_dehydrogenase, \\
\qquad\ energy = 7.1]. \\
\quad \}\}
\end{array}
$$

Figure 12: Facts of Krebs Cycle in *QUIXOTE*

Although there are many approaches to statutory interpretation, we take the following steps:

- *analogy detection*
  Given a new case, similar precedents to the case are retrieved from an existing precedent database.
- *rule transformation*
  Precedents (interpretation rules) extracted by analogy detection are abstracted until the new case can be applied to them.
- *deductive reasoning*
  Apply the new case in a deductive manner to abstract interpretation rules transformed by rule transformation. This step may include statutory application because it is used in the same manner.

Among the steps, the strategy for analogy detection is essential in legal reasoning for *more efficient* detection of *better* precedents, which decides the quality of the results of legal reasoning. As the primary objective of TRIAL at the current stage is to investigate the possibilities of *QUIXOTE* in the area and develop a prototype system, we focus only on a small target. That is, to what extent should interpretation rules be abstracted for *a* new case, in order to get an answer with a plausible explanation, but not for general abstraction mechanism.

4.2.2 TRIAL on Legal Precedent Databases

All data and knowledge in TRIAL is described in *QUIXOTE*. The system, written in KL1, is constructed on *QUIXOTE*. The overall architecture is shown in Figure 13. In the figure, *QUIXOTE* supports the functions of rule transformation (Rule Transformer) and deductive reasoning (Deductive Reasoner) as the native functions besides the database component, while TRIAL

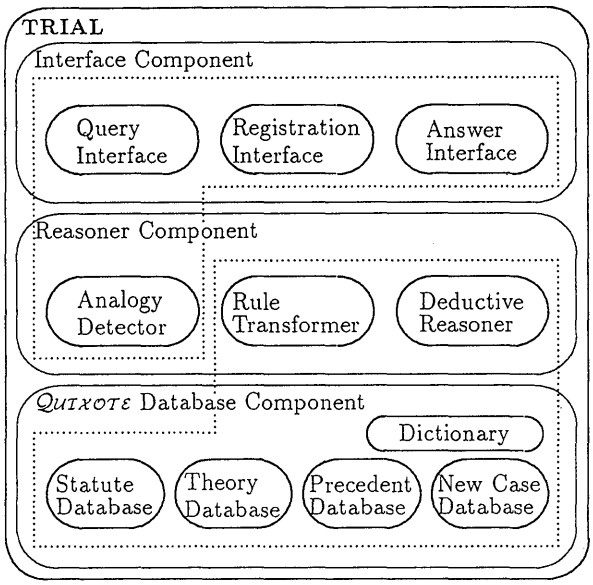


Figure 13: Architecture of TRIAL

supports the function of analogy detection (Analogy Detector) besides the interface component.

Consider a simplified example related to "*karōshi*" (death from overwork) in order to discuss the analogy detector. A new case, *new-case*, is as follows:

> *Mary*, a *driver*, employed by a company, "*S*", died from a *heart-attack* while taking a *catnap* between jobs. Can this case be applied to the *worker's compensation law*?

This is represented as a module *new-case* in QUIXOTE as follows:

$$\begin{aligned} \mathit{new\text{-}case} :: \{\{\mathit{new\text{-}case}/[&\mathit{who}=\mathit{mary},\\ &\mathit{while}=\mathit{catnap},\\ &\mathit{result}=\mathit{heart\text{-}attack}];;\\ \mathit{rel}[&\mathit{state}=\mathit{employee},\mathit{emp}=\mathit{mary}]\\ &/[\mathit{affil}=\mathit{org}[\mathit{name}=\text{“}S\text{”}],\\ &\ \mathit{job}\rightarrow\mathit{driver}]\}\} \end{aligned}$$

where ";;" is a delimiter between rules. The module is stored in the new case database. Assume that there are two *abstract* precedents [8] of *job-causality* and *job-execution*:

$$\begin{aligned} \mathit{case}_1 :: \mathit{judge}&[\mathit{case}=X]/[\mathit{judge}\rightarrow\mathit{job\text{-}causality}]\\ &\Leftarrow \mathit{rel}[\mathit{state}=Y,\mathit{emp}=Z]/[\mathit{cause}=X]\\ &\quad ||\{X\sqsubseteq \mathtt{parm}.\mathit{case},\\ &\qquad Y\sqsubseteq \mathtt{parm}.\mathit{status},\\ &\qquad Z\sqsubseteq \mathtt{parm}.\mathit{emp}\};;\\ \mathit{case}_2 :: \mathit{judge}&[\mathit{case}=X]/[\mathit{judge}\rightarrow\mathit{job\text{-}execution}]\\ &\Leftarrow X/[\mathit{while}=Y,\mathit{result}=Z],\\ &\quad Y\sqsubseteq \mathit{job}\\ &\quad ||\{X\sqsubseteq \mathtt{parm}.\mathit{case},\\ &\qquad Y\sqsubseteq \mathtt{parm}.\mathit{while},\\ &\qquad Z\sqsubseteq \mathtt{parm}.\mathit{result}\}. \end{aligned}$$

Note that variables X, Y, and Z in both rules are restricted by the properties of an object parm. That is, they are already abstracted by parm and their abstract level is controlled by parm's properties. Such precedents are retrieved from the precedent database by analogy detection and abstracted by rule transformation. We must consider the *labor-law* (in the statute database) and a *theory* (in the theory database) as follows:

$$\begin{aligned} \mathit{labor\text{-}law} :: \mathit{org}&[\mathit{name}=X]\\ &/[\mathit{resp}\rightarrow\mathit{compensation}[\mathit{obj}=Y,\\ &\qquad\qquad \mathit{money}=\mathit{salary}]]\\ &\Leftarrow \mathit{judge}[\mathit{case}\rightarrow\mathit{case}]\\ &\qquad /[\mathit{who}=Y,\\ &\qquad\ \ \mathit{result}\rightarrow\mathit{disease},\\ &\qquad\ \ \mathit{judge}\rightarrow\mathit{insurance}],\\ &\quad \mathit{rel}[\mathit{state}=Z,\mathit{emp}=Y]\\ &\qquad /[\mathit{affil}=\mathit{org}[\mathit{name}=X]]. \end{aligned}$$

$$\begin{aligned} \mathit{theory} :: \mathit{judge}&[\mathit{case}=X]/[\mathit{judge}\rightarrow\mathit{insurance}]\\ &\Leftarrow \mathit{judge}[\mathit{case}=X]/[\mathit{judge}\rightarrow\mathit{job\text{-}causality}],\\ &\quad \mathit{judge}[\mathit{case}=X]/[\mathit{judge}\rightarrow\mathit{job\text{-}execution}]\\ &\quad ||\{X\sqsubseteq \mathit{case}\}. \end{aligned}$$

Furthermore, we must define the parm object as follows:

$$\begin{aligned} \mathit{parm} :: \mathtt{parm}/[&\mathit{case}=\mathit{case},\\ &\mathit{state}=\mathit{rel},\\ &\mathit{while}=\mathit{job},\\ &\mathit{result}=\mathit{disease},\\ &\mathit{emp}=\mathit{person}]. \end{aligned}$$

In order to use parm for case_1 and case_2, we define the following submodule relation:

$$\mathit{parm}\sqsupseteq_S \mathit{case}_1\cup\mathit{case}_2.$$

This information is dynamically defined during rule transformation. Furthermore, we must define the subsumption relation:

$$\begin{array}{lcl} \mathit{case} & \sqsupseteq & \mathit{new\text{-}case}\\ \mathit{rel} & \sqsupseteq & \mathit{employee}\\ \mathit{disease} & \sqsupseteq & \mathit{heart\text{-}attack}\\ \mathit{job} & \sqsupseteq & \mathit{catnap}\\ \mathit{person} & \sqsupseteq & \mathit{mary}\\ \mathit{job\text{-}causality} & \sqsupseteq & \mathit{insurance}\\ \mathit{job\text{-}execution} & \sqsupseteq & \mathit{insurance} \end{array}$$

[8] In this paper, we omit the rule transformation step and assume that abstract interpretation rules are given.

Such definitions are stored in the dictionary in advance.

Then, we can ask some questions with a hypothesis to the above database:

1) If *new-case* inherits *parm* and *theory*, then what kind of *judgment* can we get?

$$?\text{-}new\text{-}case : judge[case{=}new\text{-}case]/[judge{=}X]$$
$$\texttt{if } new\text{-}case \sqsupseteq_S parm \cup theory.$$

we can get three answers:

- $X = job\text{-}execution$
- if $new\text{-}case : judge[case = new\text{-}case]$ has a property $judge \sqsubseteq job\text{-}causality$, then $X \sqsubseteq insurance$
- if $new\text{-}case : rel[state = employee, emp = mary]$ has a property $cause = new\text{-}case$, then $X \sqsubseteq insurance$

Two of these are answers with assumptions.

2) If *new-case* inherits *labor-law* and *parm*, then what kind of responsibility should the organization which Mary is affiliated to have?

$$?\text{-}new\text{-}case : org[name{=}\text{“}S\text{”}]/[resp{=}X]$$
$$\texttt{if } new\text{-}case \sqsupseteq_S parm \cup labor\text{-}law.$$

we can get two answers:

- if $new\text{-}case : judge[case = new\text{-}case]$ has a property $judge \sqsubseteq job\text{-}causality$, then $X \sqsubseteq compensation[obj = mary, money = salary]$
- if $new\text{-}case : rel[state = employee, emp = mary]$ has a property $cause = new\text{-}case$, then $X \sqsubseteq compensation[obj = mary, money = salary]$

For analogy detection, the parm object plays an essential role in determining how to abstract rules as in $case_1$ and $case_2$, what properties to be abstracted in parm, and what values to be set in properties of parm. In TRIAL, we have experimented with such abstraction, that is, analogy detection, in *Quixote*.

For the user interface of TRIAL, *Quixote* returns explanations (derivation graphs) with corresponding answers, if necessary. The TRIAL interface shows this graphically according to the user's request. By judging an answer from the validity of the assumptions and the corresponding explanation, the user can update the database or change the abstraction strategy.

4.3 Temporal Inference

Temporal information plays an important role in natural language processing. A time axis in natural language is, however, not homogeneous as in natural science but is relative to the events in mind: shrunken in parts and stretched in others. Furthermore, the relativity is different depending on the observer's perspective. This work aims to show the paradigm of an inference system that merges temporal information extracted from each lexical item and resolves any temporal ambiguity that a word may have [9].

4.3.1 Temporal Information in Natural Language

We can, frequently, make different expressions for the same real situation. For example,

Don Quixote attacks a windmill.
Don Quixote attacked a windmill.
Don Quixote is attacking a windmill.
⋮

Such different expressions are related to tense and aspects. How should we describe the relation between them?

According to situation theory, we write a *support* relation between a *situation* s and an *infon* σ as follows:

$$s \models \sigma.$$

For example, if one of the above examples is supported in a situation s, it is written as follows:

$$s \models \ll attack, \text{Don Quixote}, windmill \gg,$$

where *attack* is a relation, and "Don Quixote" and *windmill* are parameters. However, strictly speaking, as such a relation is cut out from a prespective $\mathcal{P}$, we should write it as follows:

$$s \models \sigma \iff \mathcal{P}(s' \models \sigma').$$

Although we might nest perspectives on such a relation, we assume some reflective property:

$$\mathcal{P}(s' \models \sigma') \implies \mathcal{P}(s')\mathcal{P}(\models)\mathcal{P}(\sigma').$$

In order to consider how to represent $\mathcal{P}(s')$ and $\mathcal{P}(\sigma')$ from a temporal point of view, we introduce a partial order relation among sets of time points. Assume that a set of time points are partially ordered by $\preceq$, then we can define $\preceq_t$ and $\subseteq$ among sets T_1 and T_2 as follows:

$$T_1 \preceq_t T_2 \stackrel{def}{=} \forall t_1 \in T_1, \forall t_2 \in T_2.\ t_1 \preceq t_2.$$
$$T_1 \subseteq T_2 \stackrel{def}{=} \forall t_1 \in T_1.\ t_1 \in T_2.$$

We omit the subscript t if there is no confusion.

In order to make tense and aspects clearer, we introduce the following concepts:

[9] See the details in [Tojo and Yasukawa 1992].

1) discrimination of an *utterance situation* u and a *described situation* s, and

2) *duration* (a set of linear time points, decided by a start point and an end point) of situations and an infon. The duration of T is written as $\| T \|_t$.

We can see the relation among three durations of an utterance situation, a described situation, and an infon in Figure 14. If there is no confusion, we use a simple

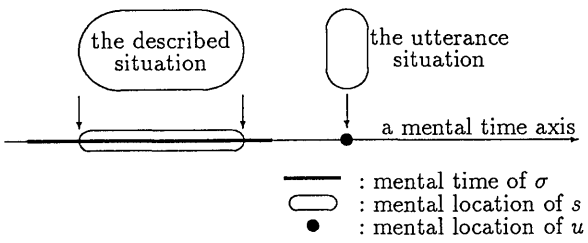


Figure 14: Relation of Three Durations

notation: $s_1 \preceq s_2$ instead of $\| s_1 \|_t \preceq \| s_2 \|_t$ and $s_1 \subseteq s_2$ instead of $\| s_1 \|_t \subseteq \| s_2 \|_t$.

By the above definitions, we can define tense and aspects when $s \models \sigma$ as follows ($\mathcal{P}(\models)$ is written as $\models$):

$$\begin{aligned} s[s \preceq u] &\models \ll past, \sigma \gg . \\ s[s \supset u] &\models \ll present, \sigma \gg . \\ s[s \subset u] &\models \ll progressive, \sigma \gg . \\ s[\sigma \preceq u] &\models \ll perfect, \sigma \gg . \end{aligned}$$

where s is a described situation, u is an utterance situation, and σ is an infon. C in $s[C]$ is a constraint, which is intended to be a perspective. The above rules are built-in rules (or axioms) for temporal inference in *Quixote*.

4.3.2 Temporal Inference in *Quixote*

We define a rule for situated inference as follows:

$$s \models \sigma \Leftarrow s_1 \models \sigma_1, \cdots, s_n \models \sigma_n.$$

where $s, s_1, \cdots, s_n$ are situations with perspectives. This rule means that $s \models \sigma$ if $s_1 \models \sigma_1$, $\cdots$, and $s_n \models \sigma_n$. Such rules can be easily translated into a subclass of *Quixote* by relating a situation with perspectives to a module, an infon to an o-term, and partial order among duration to subsumption relation. However, there is one restriction: a constraint in a rule head may not include subsumption relations between o-terms, because such a relation might destroy a subsumption lattice.

A verbalized infon is represented as an o-term as follows [10]:

$$\begin{aligned} inf[v\_rel = [rel = R, \\ cls = CLS, \\ per = P], \\ args = Args], \end{aligned}$$

where v_rel takes a verb relation and $args$ takes the arguments. R is a verb, CLS is the classification, and P is a temporal situation. For example, "John is running" is written as follows:

$$\begin{aligned} inf[v\_rel = [rel = run, \\ cls = act_2, \\ pers = [fov = ip, \\ pov = pres]], \\ args = [agt = john]]. \end{aligned}$$

That is, the agent is *john*, and the verb is *run*, which is classified in act_2 (*in-progress state* or *resultant state*), and the perspective is *in-progress state* as the field of view (an oval in Figure 14) and *present* as the point of view ($\bullet$ in Figure 14).

The discourse situation which supports such a verbalized infon is represented as follows:

$$dsit[fov = ip, pov = pres, src = U],$$

where the first two arguments are the same as the above infon's *pers* and the third argument is the utterance situation.

According to the translation, we show a small example, which makes it possible to reduce temporal ambiguity in expression.

1) Given an expression $exp = E$, each morpheme is processed in order to check the temporal information:

$$\begin{aligned} &mi[u = U, exp = [\,], e = D, infon = Infon]. \\ &mi[u = U, exp = [Exp|R], e = D, infon = Infon] \\ &\quad \Leftarrow d\_cont[exp = Exp, sit = D, infon = Infon, \\ &\qquad mi[u = U, exp = R, e = D, infon = Infon]. \end{aligned}$$

Temporal information for each morpheme is intersected in D: that is, ambiguity is gradually reduced.

2) Temporal information in a pair of a discourse situation and a verbalized infon is defined by the following rules:

[10] An o-term $\mathsf{T}[l_1 = o_1, \cdots, l_n = o_2]$ can be abbreviated as $[l_1 = o_1, \cdots, l_n = o_2]$.

$d\_cont[exp=Exp,$
$\quad sit=dsit[fov=Fov, pov=Pov, src=U]$
$\quad infon=inf[v\_rel=V\_rel, args=Args]]$
$\Leftarrow dict : v[cls=CLS, rel=R, form=Exp]$
$\quad ||\{V\_rel=[rel=R, cls=CLS, pers=P]\}$

$d\_cont[exp=Exp,$
$\quad sit=dsit[fov=Fov, pov=Pov, src=U]$
$\quad infon=inf[v\_rel=V\_rel, args=Args]]$
$\Leftarrow dict : auxv[asp=ASP, form=Exp],$
$\quad map[cls=CLS, asp=ASP, fov=Fov]$
$\quad ||\{V\_rel=[rel=\_, cls=CLS, pers=P],$
$\quad P=[fov=Fov, pov=\_\}];;$

$d\_cont[exp=Exp,$
$\quad sit=dsit[fov=Fov, pov=Pov, src=U]$
$\quad infon=inf[v\_rel=V\_rel, args=Args]]$
$\Leftarrow dict : affix[pov=Pov, form=ru]$
$\quad ||\{V\_rel=[rel=\_, cls=\_, pers=P],$
$\quad P=[fov=\_, pov=Pov]\}$

3) There is a module *dict*, where lexical information is defined as follows:

$dict::\{\{$
$\quad v[cls=act_1, rel=put_on, form=ki];;$
$\quad v[cls=act_2, rel=run, form=$hashi$];;$
$\quad v[cls=act_3, rel=understand, form=$waka$];;$
$\quad auxv[asp=state, form=tei];;$
$\quad affix[pov=pres, form=ru];;$
$\quad affix[pov=past, form=ru]\}\}$

where *form* has a value of Japanese expression. Further, mapping of field of view is also defined as a set of (global) facts as follows:

$map[cls=act1, asp=state, fov=\{ip, tar, res\}].$
$map[cls=act2, asp=state, fov=\{ip, res\}].$
$map[cls=act3, asp=state, fov=\{tar, res\}].$

If some Japanese expression is given in a query, the corresponding temporal information is returned by the above program.

5 Towards More Flexible Systems

In order to extend a DOOD system, we take other approaches for more flexible execution control, mainly focusing on natural language applications as its examples.

5.1 Constraint Transformation

There are many natural language grammar theories: transformational and constraint-base grammar such as GB, unification-based and rule-based grammar such as GPSG and LFG, and unification-based and constraint-based grammar such as HPSG and JPSG. Considering a more general framework of grammar in logic programming, HPSG and JPSG are considered to be better, because morphology, syntax, semantics, and pragmatics are uniformly treated as constraints. From such a point of view, we developed a new constraint logic programming (CLP) language, *cu-Prolog*, and implemented a JPSG (Japanese Phrase Structure Grammar) parser in it [11].

5.1.1 Constraints in Unification-Based Grammar

First, consider various types of constraints in constraint-based grammar:

- A *disjunctive feature structure* is used as a basic information structure, defined like nested tuples or complex objects as follows:

  1) A *feature structure* is a tuple consisting of pairs of a label and a value: $[l_1=v_1, \cdots, l_n=v_n]$.

  2) A *value* is an atom, a feature structure, or a set $\{f_1, \cdots, f_n\}$ of feature structures.

- In JPSG, grammar rules are described in the form of a binary tree as in Figure 15, each node of which is a feature structure: in which a specific

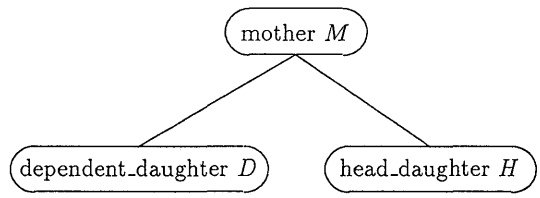


Figure 15: Phrase Structure in JPSG

feature (attribute) decides whether D works as a complement or as a modifier. Note that each grammar, called a *structural principle*, is expressed as the constraints among three features, M, D, and H, in the local phrase structure tree.

As shown in the above definition, feature structures are very similar to the data structure in DOOD [12]. We will see some requirements of natural language processing for our DOOD system and develop applications on the DOOD system.

[11] See the details in [Tsuda 1992].

[12] This is one of the reason why we decided to design *QUIXOTE*. See the appendix.

5.1.2 cu-Prolog

In order to process feature structures efficiently, we have developed a new CLP called *cu-Prolog*. A rule is defined as follows [13]:

$$H \Leftarrow B_1, \cdots, B_n || C_1, \cdots, C_m.$$

where $H, B_1, \cdots, B_n$ are atomic formulas, whose arguments can be in the form of feature structures and $C_1, \cdots, C_m$ are constraints in the form of an equation among feature structures, variables, and atoms, or an atomic formula defined by another set of rules. There is a restriction for an atomic formula in constraints in order to guarantee the congruence of constraint solving. This can be statically checked. The semantic domain is a set of relations of partially tagged trees, as in CIL[Mukai 1988] and the constraint domain is also the same.

The derivation in cu-Prolog is a sequence of a pair (G, C) of a set of subgoals and a set of constraints, just as in conventional CLP. Their differences are as follows:

- All arguments in predicates can be feature structures, that is, unification between feature structures is necessary.
- A computation rule does not select a rule which does not contribute to constraint solving: in the case of $(\{A\} \cup G, C)$, $A' \Leftarrow B || C'$, and $A\theta = A'\theta$, the rule is not selected if a new constraint $C\theta \cup C'\theta$ cannot be reduced.
- The constraint solver is based on unfold/fold transformation, which produces new predicates dynamically in a constraint part.

'Disjunction' in feature structures of cu-Prolog is treated basically as 'conjunction', just as in an o-term in $\mathcal{QUIXOTE}$ and a nested term in Kappa (CRL). However, due to the existence of a predicate, disjunction is resolved (or unnested) by introducing new constraints and facts:

$$H \Leftarrow p([l=\{a,b\}]) \iff \begin{array}{l} H \Leftarrow p([l=X]) || \{new\_p(X)\}. \\ new\_p(a). \\ new\_p(b). \end{array}$$

That is, in cu-Prolog, disjunctive feature structures are processed in OR-parallel, in order to avoid set unification as in CRL. Only by focusing on the point does the efficiency seem to depend on whether we want to obtain all solutions or not.

One of the distinguished features in cu-Prolog is dynamic unfold/fold transformation during query processing, which contributes much to improving the efficiency of query processing. Some examples of a JPSG parser in cu-Prolog appear in [Tsuda 1992]. As predicate-based notation is not essential, language features in cu-Prolog can be encoded into the specification of $\mathcal{QUIXOTE}$ and the constraint solver can also be embedded into the implementation of $\mathcal{QUIXOTE}$ without changing semantics.

[13] As we are following with the syntax of $\mathcal{QUIXOTE}$, the following notation is different from cu-Prolog.

5.2 Dynamical Programming

This work aims to extend a framework of constraint throughout computer and cognitive sciences [14]. In some sense, the idea originates in the treatment of constraints in cu-Prolog. Here, we describe an outline of dynamical programming as a general framework of treating constraints and an example in natural language processing.

5.2.1 Dynamics of Symbol Systems

As already mentioned in Section 2, partial information plays an essential role in knowledge information processing systems. So, knowing how to deal with the partiality will be essential for future symbol systems. We employ a constraint system, which is independent of information flow. In order to make the system computationally more tractable than conventional logic, it postulates a *dynamics* of constraints, where the state of the system is captured in terms of *potential energy*.

Consider the following program in the form of clauses:

$$p(X) \Leftarrow r(X,Y), p(Y).$$
$$r(X,Y) \Leftarrow q(X).$$

Given a query $?\text{-}p(A), q(B)$, the rule-goal graph as used in deductive databases emulates top-down evaluation as in Figure 16. However, the graph presupposes a cer-

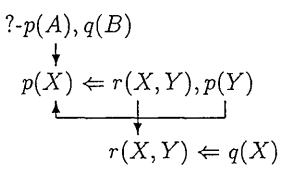


Figure 16: Rule-Goal Graph

tain information flow such as top-down or bottom-up evaluation. More generally, we consider it in the form in Figure 17. where the lines represent (partial) equations among variables, and differences between variables are not written for simplicity. We call such a graph a *constraint network*.

In this framework, computation proceeds by propagating constraints in a node (a variable or an atomic

[14] See the details in [Hasida 1992].

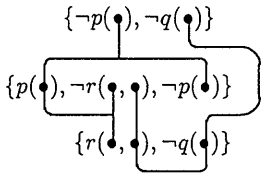


Figure 17: Constraint network

constraint) to others on the constraint network. In order to make such computation possible, we note the dynamics of constraints, as outlined below:

1) An *activation value* is assigned to each atomic constraint (an atomic formula or an equation). The value is a real number between 0 and 1 and is considered as the truth value of the constraint.

2) Based on activation values, *normalization energy* is defined for each atomic constraint, *deduction energy* and *abduction energy* are defined for each clause, and *assimilation energy* and *completion energy* are defined for possible unifications. The *potential energy* U is the sum of the above energies.

3) If the current state of a constraint is represented in terms of a point x of Euclidean space, U defines a *field of force* F of the point x. F causes *spreading activation* when $F \neq 0$. A change of x is propagated to neighboring parts of the constraint network, in order to reduce U. In the long run, the assignment of the activation values settles upon a stable equilibrium satisfying $F = 0$.

Symbolic computation is also controlled on the basis of the same dynamics. This computational framework is not restricted in the form of Horn clauses.

5.2.2 Integrated Architecture of Natural Language Processing

In traditional natural language processing, the system is typically a sequence of syntactic analysis, semantic analysis, pragmatic analysis, extralinguistic inference, generation planning, surface generation, and so on. However, syntactic analysis does not necessarily precede semantic and pragmatic comprehension, and generation planning is entwined with surface generation. Integrated architecture is expected to remedy such a fixed information flow. Our dynamics of constraint is appropriate for such an architecture.

Consider the following example:

Tom took a telescope. He saw a girl with it.

We assume that *he* and *it* are anaphoric with *Tom* and *the telescope*, respectively. However, *with it* has attachment ambiguity:

Tom has a telescope when he sees the girl, or
the girl has the telescope when Tom sees her.

Consider a set of facts:

(1) $take(tom, telescope)$.

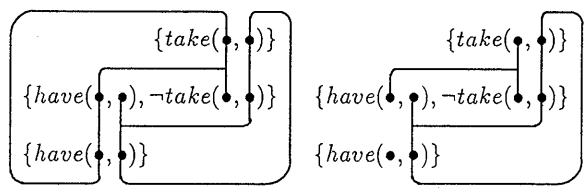
(2) $have(tom, telescope)$.

(3) $have(girl, telescope)$.

and an inference rule:

(4) $have(X, Y) \Leftarrow take(X, Y)$.

By constructing the constraint networks of (1),(2),(4) and (1),(3),(4) as in Figure 18, we can see that there



Constraint Network of (2) Constraint Network of (3)

Figure 18: Constraint Networks of Alternatives

are two cycles (involving *tom* and *telescope*) in the left network ((1), (2), and (4)), while there is only one cycle (*girl*) in the right network ((1), (3), and (4)). From the viewpoint of potential energy, the former tends to excite more strongly than the latter, in other words, (2) is more plausible than (3).

Although, in natural language processing, resolution of ambiguity is a key point, the traditional architecture has not been promising, while our integrated architecture based on a dynamics of constraint network seems to give more possibilities not only for such applications but also for knowledge-base management systems.

6 Related Works

Our database and knowledge-base management system in the framework of DOOD has many distinguished features in concept, size, and varieties, in comparison with other systems. The system aims not only to propose a new paradigm but also to provide database and knowledge-base facilities in practice for many knowledge information processing systems.

There are many works, related to DOOD concepts, for embedding object-oriented concepts into logic programming. Although *F-logic*[Kifer and Lausen 1989] has the richest concepts, the *id-term* for object identity is based on predicate-based notation and properties are insufficient from a constraint point of view. Furthermore, it lacks update functions and a module concept.

QUIXOTE has many more functions than F-logic. Although, in some sense, *QUIXOTE* might be an over-specification language, users can select any subclass of *QUIXOTE*. For example, if they use only a subclass of object terms, they can only be conscious of the sub-language as a simple extension of Prolog.

As for nested relational models, there are many works since the proposal in 1977, and several models have been implemented: *Verso* [Verso 1986], *DASDBS* [Schek and Weikum 1986], and *AIM-P* [Dadam *et al.* 1986]. However, the semantics of our model is different from theirs. As the (extended) NF^2 model of DASDBS and AIM-P has set-based (higher order) semantics, it is very difficult to extend the query capability *efficiently*, although the semantics is intuitively familiar to the user. On the other hand, as Verso is based on the universal relation schema assumption, it guarantees efficient procedural semantics. However, the semantics is intuitively unfamiliar to the user: even if $t \notin \sigma_1 T$ and $t \notin \sigma_2 T$ for a relation T, it might happen that $t \in \sigma_1 T \cup \sigma_2 T$. Compared with them, Kappa takes simple semantics, as mentioned in Section 3. This semantics is retained in o-terms in *QUIXOTE* and disjunctive feature structures in cu-Prolog for efficient computation.

As for genetic information processing, researchers in logic programming and deductive databases have begun to focus on this area as a promising application. However, most of these works are devoted to query capabilities such as transitive closure and prototyping capabilities, while there are few works which focus on data and knowledge representation. On the other hand, *QUIXOTE* aims at both the above targets. As for legal reasoning, there are many works based on logic programming and its extensions. Our work has not taken their functions into consideration, but has reconsidered them from a database point of view, especially by introducing a module concept.

7 Future Plans and Concluding Remarks

We have left off some functions due to a shortage in man power and implementation period. We are considering further extensions through the experiences of our activities, as mentioned in this paper.

First, as for *QUIXOTE*, we are considering the following improvements and extensions:

- Query transformation techniques such as sideways information passing and partial evaluation are not fully applied in the current implementation. Such optimization techniques should be embedded in *QUIXOTE*, although constraint logic programming needs different devices from conventional deductive databases. Furthermore, for more efficient query processing, flexible control mechanisms, such as in cu-Prolog and dynamical programming, would be embedded.

- For more convenience for description in *QUIXOTE*, we consider meta-functions as HiLog [Chen *et al.* 1989]:

  $$tc(R)(X,Y) \text{ :- } R(X,Y)$$
  $$tc(R)(X,Y) \text{ :- } tc(R)(X,Z), tc(R)(Z,Y)$$

  In order to provide such a function, we must introduce new variables ranging over basic objects.

  This idea is further extended to a platform language of *QUIXOTE*. For example, although we must decide the order relation (such as Hoare, Smyth, or Egli-Milner) among sets in order to introduce a set concept, the decision seems to depend on the applications. For more applications, such a relation would best be defined by a platform language. The current *QUIXOTE* would be a member of a family defined in such a platform language.

- Communication among *QUIXOTE* databases plays an important role not only for distributed knowledge-bases but also to support *persistent view*, *persistent hypothesis*, and *local* or *private* databases. Furthermore, cooperative query processing among agents defined *QUIXOTE* is also considered, although it closely depends on the ontology of object identity.

- In the current implementation, *QUIXOTE* objects can also be defined in KL1. As it is difficult to describe every phenomena in a single language, as you know, all languages should support interfaces to other languages. Thus, in *QUIXOTE* too, a multi-language system would be expected.

- Although, in the framework of DOOD, we have focused mainly on data modeling extensions, the direction is not necessarily orthogonal from logical extensions and computational modeling extensions: set grouping can emulate negation as failure and the procedural semantics of *QUIXOTE* can be defined under the framework of object-orientation.

  However, from the viewpoint of artificial intelligence, non-monotonic reasoning and 'fuzzy' logic should be further embedded, and, from the viewpoint of design engineering, other semantics such as object-orientation, should also be given.

As for Kappa, we are considering the following improvements and extensions:

- In comparison with other DBMSs by Wisconsin Benchmark, the performance of Kappa can be further improved, especially in extended relational

algebra, by reducing inter-kernel communication costs. This should be pursued separately from the objective.

- It is planned for Kappa to be accessed not only from sequential and parallel inference machines but also from general purpose machines or workstations. Furthermore, we should consider the portability of the system and the adaptability for an open system environment. One of the candidates is heterogeneous distributed DBMSs based on a client-server model, although Kappa-P is already a kind of distributed DBMS.

- In order to provide Kappa with more applications, customizing facilities and service utilities should be strengthened as well as increasing compatibility with other DBMSs.

In order to make Kappa and *Quixote* into an integrated knowledge-base management system, further extensions are necessary:

- *Quixote* takes nested transaction logic, while Kappa takes flat transaction logic. As a result, *Quixote* guarantees persistence only at the top level transaction. In order to couple them more tightly, Kappa should support nested transaction logic.

- From the viewpoint of efficient processing, users cannot use Kappa directly through *Quixote*. This, however, causes difficulty with object identity, because Kappa does not have a concept of object identity. A mechanism to allow Kappa and *Quixote* to share the same object space should be considered.

- Although Kappa-P is a naturally parallel DBMS, current *Quixote* is not necessarily familiar with parallel processing, even though it is implemented in KL1 and works in parallel. For more efficient processing, we must investigate parallel processing in Kappa and *Quixote*.

We must develop bigger applications than those we mentioned in this paper. Furthermore, we must increase the compatibility with the conventional systems: for example, from Prolog to *Quixote* and from the relational model to our nested relational model.

We proposed a framework for DOOD, and are engaged in various R&D activities for databases and knowledge-bases in the framework, as mentioned in this paper. Though each theme does not necessarily originate from the framework, our experiences indicate that this direction is promising for many applications.

Acknowledgments

The authors have had much cooperation from all members of the third research laboratory of ICOT for each topic. We especially wish to thank the following people for their help in the specified topics: Hiroshi Tsuda for *Quixote* and *cu-Prolog*, Moto Kawamura and Kazutomo Naganuma for *Kappa*, Hidetoshi Tanaka and Yuikihiro Abiru for *Biological Databases*, Nobuichiro Yamamoto for *TRIAL*, Satoshi Tojo for *Temporal Inference*, and Kôiti Hasida for *DP*.

We are grateful to members of the DOOD (DBPL, ETR, DDB&AI, NDB, IDB), STASS, and JPSG working groups for stimulating discussions and useful comments on our activities, and, not to mention, all members of the related projects (see the appendix) for their implementation efforts.

We would also like to acknowledge Kazuhiro Fuchi and Shunichi Uchida without whose encouragement *Quixote* and Kappa would not have been implemented.

References

[Aczel 1988] P. Aczel, *Non-Well Founded Set Theory*, CSLI Lecture notes No. 14, 1988.

[Chen *et al.* 1989] W. Chen, M. Kifer and D.S. Warren, "HiLog as a Platform for Database Language", *Proc. the Second Int. Workshop on Database Programming Language*, pp.121-135, Gleneden Beach, Oregon, June, 1989.

[Chikayama 1984] T. Chikayama, "Unique Features of ESP", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, Nov.6-9, 1984.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato, and T. Miyazaki,"Overview of the Parallel Inference Machine Operating Sistem (PIMOS)", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, Nov.28-Dec.2, 1988.

[Dadam *et al.* 1986] P. Dadam, et al, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies", *ACM SIGMOD Int. Conf. on Management of Data*, 1986.

[Delobel *et al.* 1991] C. Delobel, M. Kifer, and Y. Masunaga (eds.), *Deductive and Object-Oriented Databases*, (*Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases (DOOD'91)*), LNCS 566, Springer, 1991.

[Goto *et al.* 1988] A. Goto *et al.*,"Overview of the Parallel Inference Machine Architecture (PIM)", *Proc.*

Int. Conf. on Fifth Generation Computer Systems, ICOT, Tokyo, Nov.28-Dec.2, 1988.

[Haniuda *et al.* 1991] H. Haniuda, Y. Abiru, and N. Miyazaki, "PHI: A Deductive Database System", *Proc. IEEE Pacific Rim Conf. on Communication, Computers, and Signal Processing*, May, 1991.

[Hasida 1992] K. Hasida, "Dynamics of Symbol Systems — An Integrated Architecture of Cognition", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Kawamura *et al.* 1992] M. Kawamura, H. Naganuma, H. Sato, and K. Yokota, "Parallel Database Management System *Kappa-P*", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Kifer and Lausen 1989] M. Kifer and G. Lausen, "F-Logic — A Higher Order Language for Reasoning about Objects, Inheritance, and Schema", *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp.134-146, Portland, June, 1989.

[Kim *et al.* 1990] W. Kim, J.-M. Nicolas, and S. Nishio (eds.), *Deductive and Object-Oriented Databases*, (*Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD89)*), North-Holland, 1990.

[Miyazaki *et al.* 1989] N. Miyazaki, H. Haniuda, K. Yokota, and H. Itoh, "A Framework for Query Transformation", *Journal of Information Processing*, vol.12, No.4, 1989.

[Mukai 1988] K. Mukai, "Partially Specified Term in Logic Programming for Linguistic Analysis", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, Nov.28-Dec.2, 1988.

[Schek and Weikum 1986] H.-J. Schek and G. Weikum, "DASDBS: Concepts and Architecture of a Database System for Advanced Applications", *Tech. Univ. of Darmstadt, Technical Report*, DVSI-1986-T1, 1986.

[Tanaka 1992] H. Tanaka, "Integrated System for Protein Information Processing", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Tojo and Yasukawa 1992] S. Tojo and H. Yasukawa, "Situated Inference of Temporal Information", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Tsuda 1992] H. Tsuda, "cu-Prolog for Constraint-Based Grammar", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Ueda and Chikayama 1990]
K. Ueda and T. Chikayama, "Design of the Kernel Language for thr Parallel Ingerence Machine", *The Computer Journal*, vol.33, no.6, 1990.

[Verso 1986] J. Verso, "VERSO: A Data Base Machine Based on Non 1NF Relations", *INRIA Technical Report*, 523, 1986.

[Yamamoto 1990] N. Yamamoto, "TRIAL: a Legal Reasoning System (Extended Abstract)", *Joint French-Japanese Workshop on Logic Programming*, Renne, France, July, 1991.

[Yasukawa *et al.* 1992] H. Yasukawa, H. Tsuda, and K. Yokota, "Object, Properties, and Modules in *Quixote*", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, June 1-5, 1992.

[Yokota 1988] K. Yokota, "Deductive Approach for Nested Relations", *Programming of Future Generation Computers II*, eds. by K. Fuchi and L. Kott, North-Holland, 1988.

[Yokota *et al.* 1988] K. Yokota, M. Kawamura, and A. Kanaegami, "Overview of the Knowledge Base Management System (KAPPA)", *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, Nov.28-Dec.2, 1988.

[Yokota and Nishio 1989] K. Yokota and S. Nishio, "Towards Integration of Deductive Databases and Object-Oriented Databases — A Limited Survey", *Proc. Advanced Database System Symposium*, Kyoto, Dec., 1989.

[Yoshida 1991] K. Yoshida, "The Design Principle of the Human Chromosome 21 Mapping Knowledgebase (Version CSH91)", *Inetrnal Technical Report of Lawrence Berkley Laboratory*, May, 1991.

Appendix

Notes on Projects for Database and Knowledge-Base Management Systems

In this appendix, we describe an outline of projects on database and knowledge-base management systems in the FGCS project. A brief history is shown in Figure 19 [15]. Among these projects, Mitsubishi Electric Corp. has cooperated in Kappa-I, Kappa-II, Kappa-P, DO-l, CIL, and *Quixote* projects, Oki Electric Industry Co., Ltd. has cooperated in PHI (DO-ϕ) and *Quixote* projects, and Hitachi, Ltd. has cooperated in ETA (DO-η) and *Quixote* projects.

a. Kappa Projects

In order to provide database facilities for knowledge information processing systems, a *Kappa* [16] project begun in September, 1985 (near the beginning of the intermediate stage of the FGCS project). The first target was to build a database with electronic dictionaries including concept taxonomy for natural language processing systems and a database for mathematical knowledge for a proof checking system called CAP-LA. The former database was particularly important: each dictionary has a few hundred thousands entries, each of which has a complex data structure. We considered that the normal relational model could not cope with such data and decided to adopt a nested relational model. Furthermore, we decided to add a new type *term* for handling mathematical knowledge. The DBMS had to be written in ESP and work on PSI machines and under the SIMPOS operating system. As we were afraid of whether the system in ESP would work efficiently or not, we decided on the semantics of a nested relation and started to develop a prototype system called *Kappa-I*. The system, consisting of 60 thousands lines in ESP, was completed in the spring of 1987 and was shown to work efficiently for a large amount of dictionary data. The project was completed in August, 1987 after necessary measurement of the processing performance.

After we obtained the prospect of efficient DBMS on PSI machines, we started the next project, *Kappa-II*[Yokota *et al.* 1988] in April, 1987, which aims at a practical DBMS based on the nested relational model. Besides the objective of more efficient performance than *Kappa-I*, several improvements were planned: a main memory database facility, extended relational algebra, user-definable command facility, and user-friendly window interface. The system, consisting of 180 thousand lines in ESP, works 10 times more efficiently in PSI-II machines than Kappa-I does in PSI-I. The project was over in March, 1989 and the system was widely released, not only for domestic organizations but also for foreign ones, and mainly for genetic information processing.

To handle larger amounts of data, a parallel DBMS project called *Kappa-P*[Kawamura *et al.* 1992] was started in February, 1989. The system is written in KL1 and works under an environment of PIM machines and the PIMOS operating system. As each local DBMS of Kappa-P works on a single processor with almost the same efficiency as Kappa-II, the system is expected to work on PIM more efficiently than *Kappa-II*, although their environments are different.

b. Deductive Database Projects

There were three projects for deductive databases.

First, in parallel with the development of Kappa, we started a deductive database project called *CRL* (complex record language) [Yokota 1988], which is a logic programming language newly designed for treating nested relations.

CRL is based on a subclass of complex objects constructed by set and tuple constructors and with a *module* concept. The project started in the summer of 1988 and the system, called *DO-l*, was completed in November, 1989. The system works on *Kappa-II*. The query processing strategy is based on methods of generalized magic sets and semi-naive evaluation. In it, rule inheritance among modules based on submodule relations are dynamically evaluated.

Secondly, we started a project called *PHI* [Haniuda *et al.* 1991] in the beginning of the intermediate stage (April, 1985). This aimed at more efficient query processing in traditional deductive databases than other systems. The strategy is based on three kinds of query transformation called *Horn clause transformation* (*HCT*)[Miyazaki *et al.* 1989]: HCT/P executes partial evaluation or unfolding, HCT/S propagates binding information without rule transformation, and HCT/R transforms a set of rules in order to restrict the search space and adds related new rules. The HCT/R corresponds to the generalized magic set strategy. By combining these strategies, PHI aims at more efficient query processing. The consequent project is called *DO-ϕ*, in which we aim at a deductive mechanism for complex objects.

Thirdly, we started a project called *ETA* in April, 1988, which aimed at knowledge-base systems based on knowledge representation such as semantic networks. One year later, the project turned towards extensions of deductive databases and was called *DO-η*.

[15] At the initial stage of the FGCS project, there were other projects for databases and knowledge-based: *Delta* and *Kaiser*, however these were used for targets other than databases and knowledge-bases.

[16] A term *Kappa* stands for *knowledge application oriented advanced database management system.*

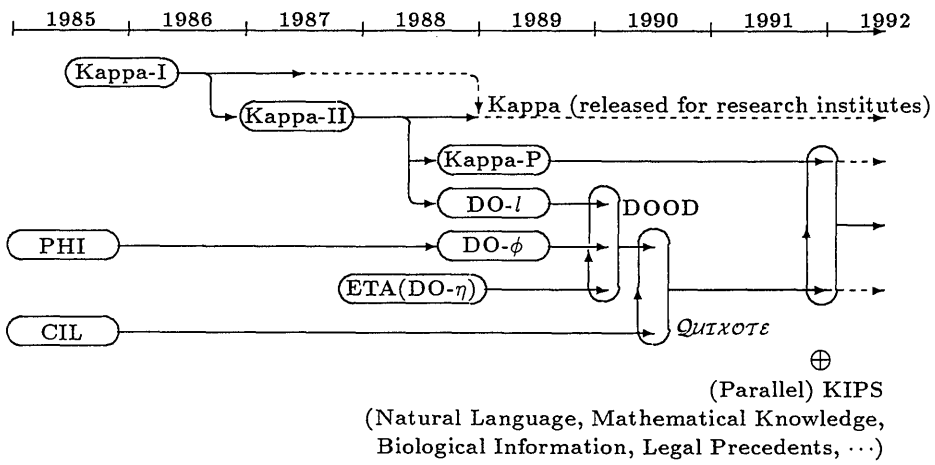


Figure 19: Brief History of Projects on Database and Knowledge-Base Management Systems

"*DO*" in the above projects stands for deductive and object-oriented databases and is shown to adopt a concept of DOODs [Yokota and Nishio 1989] as its common framework.

c. CIL Project

A language called *CIL* (complex indeterminates language) was proposed in April, 1985 [Mukai 1988]. The language aimed at semantic representation in natural language processing and was used not only in the discourse understanding system called *DUALS*, but also for representing various linguistic information. The implementation of CIL was improved several times and CIL was released to many researchers in natural language processing. The language is a kind of constraint logic programming and closely relates to situation theory and semantics. The language is based on *partially specified terms*, each of which is built by a tuple constructor. A set constructor was introduced into partially specified terms in another language *cu-Prolog*, as mentioned in Section 5.1.

d. *Quixote* Project

We tried to extend CRL not only for nested relations but also for DOODs, and to extend CIL for more efficient representation, such as the disjunctive feature structure. After these efforts, we proposed two new languages: *Juan*, as an extension of CRL, and *QUINT*, as an extension of CIL. While designing their specifications, we found many similarities between *Juan* and *QUINT*, and between concepts in databases and natural language processing, and decided to integrate these languages. The integrated language is *Quixote*[Yasukawa *et al.* 1992] (with Spanish pronunciation) [17]. As the result of integration, *Quixote* has various features, as mentioned in this paper. The *Quixote* project was started in August, 1990. The first version of *Quixote* was released to restricted users in December, 1991, and the second version was released for more applications at the end of March, 1992. Both versions are written in KL1 and work on parallel inference machines.

e. Working Groups on DOOD and STASS

At the end of 1987, we started to consider integration of logic and object-orientation concepts in the database area. After discussions with many researchers, we formed a working group for DOOD and started to prepare a new international conference on deductive and object-oriented databases [18]. The working group had four sub-working-groups in 1990: for database programming languages (DBPL), deductive databases and artificial intelligence (DDB&AI), extended term representation (ETR), and biological databases (BioDB). In 1991, the working group was divided into intelligent databases (IDB) and next generation databases (NDB). In their periodic meetings [19], we discussed not only problems of DOOD but also directions and problems

[17] Our naming convention follows the DON series, such as Don Juan and Don Quixote, where DON stands for "Deductive Object-Oriented Nucleus".

[18] Most of the preparation up until the first international conference (DOOD89) was continued by Professor S. Nishio of Osaka University.

[19] Their chairpersons are Yuzuru Tanaka of Hokkaido U. for DOOD, Katsumi Tanaka of Kobe U. for DBPL, Chiaki Sakama of ASTEM for DDB&AI and IDB, Shojiro Nishio of Osaka U. for ETR, Akihiko Konagaya of NEC for BioDB, and Masatoshi Yoshikawa of Kyoto Sangyo U. for NDB.

of next generation databases. These discussions contributed greatly to our DOOD system.

From another point of view, we formed a working group (STS) [20] for situation theory and situation semantics in 1990. This also contributed to strengthening other aspects of *Quixote* and its applications.

[20]The chairperson is Hozumi Tanaka of Tokyo Institute of Technology.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

CONSTRAINT LOGIC PROGRAMMING SYSTEM
– CAL, GDCC AND THEIR CONSTRAINT SOLVERS –

Akira Aiba and Ryuzo Hasegawa
Fourth Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{aiba, hasegawa}@icot.or.jp

Abstract

This paper describes constraint logic programming languages, CAL (*Contrainte Avec Logique*) and GDCC (*Guarded Definite Clauses with Constraints*), developed at ICOT.

CAL is a sequential constraint logic programming language with algebraic, Boolean, set, and linear constraint solvers. GDCC is a parallel constraint logic programming language with algebraic, Boolean, linear, and integer parallel constraint solvers.

Since the algebraic constraint solver utilizes the Buchberger algorithm, the solver may return answer constraints including univariate nonlinear equations. The algebraic solvers of both CAL and GDCC have the functions to approximate the real roots of univariate equations to obtain all possible values of each variable. That is, this function gives us the situation in which a certain variable has more than one value. To deal with this situation, CAL has a multiple environment handler, and GDCC has a block structure.

We wrote several application programs in GDCC to show the feasibility of the constraint logic programming language.

1 Introduction

The Fifth Generation Computer System (FGCS) project is a Japanese national project that started in 1982. The aim of the project is to research and develop new computer technologies for knowledge and symbol processing parallel computers.

The FGCS prototype system has three layers: the prototype hardware system, the basic software system, and the knowledge programming environment. Parallel application software has been developed for these. The constraint logic programming system is one of the systems that form, together with the knowledge base construction and the programming environment, the knowledge programming environment. In this paper, we describe the overall research results of constraint logic programming systems in ICOT.

The programming paradigm of constraint logic programming (*CLP*) was proposed by A. Colmerauer [Colmerauer 1987] and J. Jaffar and J-L. Lassez [Jaffar and Lassez 1987] as an extension of logic programming by extending its computation domain. Jaffar and Lassez showed that CLP possesses logical, functional, and operational semantics which coincide with each other, in a way similar to logic programming [van Emden and Kowalski 1976].

In 1986, we began to research and develop high-level programming languages suitable for problem solving to achieve our final goal, that is, developing efficient and powerful parallel CLP languages on our parallel machine.

The descriptive power of a CLP language is strongly depend on its constraint solver, because a constraint solver determines the domain of problems which can be handled by the CLP language. Almost all existing CLP languages such as Prolog III [Colmerauer 1987] and CLP($\mathcal{R}$) [Jaffar and Lassez 1987] has a constraint solver for linear equations and linear inequalities.

Unlike the other CLP languages, we focused on nonlinear algebraic equation constraints to deal with problems which are described in terms of nonlinear equations such as handling robot problem. For the purpose, we selected the Buchberger algorithm for a constraint solver of our languages.

Besides of nonlinear algebraic equations, we were also interested in writing Boolean constraints, set constraints, linear constraints, and hierarchical constraints in our framework. For Boolean constraints, we modify the Buchberger algorithm to be able to handle Boolean constraints, and later, we developed the algorithm for Boolean constraints based on the Boolean unification. For set constraints, we expand the algorithm for Boolean constraints based on the Buchberger algorithm. We also implemented the simplex method to deal with linear equations and linear inequalities same as the other CLP languages. Furthermore, we tried to handle hierarchical constraints in our framework.

We developed two CLP language processors, first we implemented a language processor for sequential CLP

language named CAL (*Contrainte Avec Logique*) on sequential inference machine PSI, and later, we implemented a language processor for parallel CLP language named GDCC (*Guarded Definite Clauses with Constraints*), based on our experiments on extending CAL processor by introducing various functions.

In Section 2, we briefly review CLP, and in Section 3, we describe CAL. In Section 4, we describe GDCC, and in Section 5, we describe various constraint solvers and their parallelization. In Section 6, we introduce application programs written in our languages.

2 CLP and the role of the constraint solver

CAL and GDCC belong to the family of CLP languages. The concept of CLP stems from the common desire for easy programming. In fact, as claimed in the literature [Jaffar and Lassez 1987, Sakai and Aiba 1989], the CLP is a scheme of programming languages with the following outstanding features:

- Natural declarative semantics.
- Clear operational semantics that coincide with the declarative semantics.

Therefore, it gives the user a paradigm of declarative (and thus, hopefully easy) programming and gives the machine an effective mechanism for execution that coincide with the user's declaration.

For example, in Prolog (the most typical instance of CLP), we can read and write programs in declarative style like "... if ... and ...". The system execute these by a series of operations with unification as its basic mechanism.

Almost every CLP language has a similar programming style and a mechanism which plays the similar role to the unification mechanism in Prolog, and the execution of programs depends on the mechanism heavily. We call such a mechanism the constraint solver of the language.

Usually, a CLP language aims at a particular field of problems and its solver has special knowledge to solve the problems. In the case of Prolog, the problems are syntactic equalities between terms, that is, the unification. On the other hand, CAL and GDCC are tuned to deal with the following:

- algebraic equations
- Boolean equations
- set inclusion and membership
- linear inequalities

These relations are called constraints.

In the CLP paradigm, a problem is expressed as constraints on the objects in the problem. Therefore, an often cited benefit of CLP is that "One does not need to write an implementation but a specification." In other words, all that a programmer should write in CLP is constraints between the objects, but not how to find objects satisfying the relation. To be more precise, such constraints are described in the form of a logical combination of formulas each of which expresses a basic unit of the relation.

Though there are many others, the above benefit surely expresses an important feature of CLP. Building an equation is usually easier than solving it. Similarly, one may be able to write down the relation between the objects without knowing the method to find the appropriate values of objects which satisfy the relation.

An ideal CLP system should allow a programmer to write any combination of any well-formed formulas. The logic programming paradigm gives us a rich framework for handling logical combinations of constraints. However, we still need a powerful and flexible constraint solver to handle each constraint. To discuss the function of the constraint solver from a theoretical point of view, the declarative semantics of CLP [Sakai and Aiba 1989] gives us several criteria. Assume that constraints are given in the form of their conjunction. Then, the following are the criteria.

(1) Can the solver decide whether a given constraint is satisfiable?

(2) Given satisfiable constraints, is there any way for the solver to express all the solutions in simplified form?

Prolog's constraint solver, the unification algorithm, answers these criteria affirmatively and so do the solvers in CAL and GDCC. In fact, they satisfy the following stronger requirements almost perfectly:

(3) Given a set of constraints, can the solver compute the simplest form (called the canonical form of the constraints) in a certain sense?

However, these criteria may not be sufficient from an applicational point of view. For example, we may sometimes be asked the following:

(4) Given satisfiable constraints, can the solver find at least one concrete solution?

Finding a concrete solution is a question usually independent of the above and may be proved theoretically impossible to answer. Therefore, we may need an approximate solution to answer this partly. As discussed later, we incorporated many of the constraint solvers and functions into CAL and GDCC.

Another important feature of constraint solvers is their incrementality. An incremental solver can be given a constraint successively. It reduces each constraint as simple

as possible by the current set of constraints. Thus, an incremental solver finds the unsatisfiability of a set of constraints as early as possible and makes Prolog-type backtracking mechanism efficient. Fortunately, the solvers of CAL and GDCC are fully incremental like unification.

3 CAL - Sequential CLP Language

This section summarizes the syntax of CAL. For a detailed description of CAL syntax, refer to the CAL User's Manual [CAL Manual].

3.1 CAL language

The syntax of CAL is similar to that of Prolog, except for its constraints. A CAL program features two types of variables: logical variables denoted by a sequence of alphanumeric characters starting with an uppercase letter (as with Prolog variables), and constraint variables denoted by a sequence of alphanumeric characters starting with a lowercase letter. Constraint variables are global variables, while logical variables are local variables within the clauses in which they occur. This distinction is introduced to simplify incremental querying.

The following is an example CAL program that features algebraic constraints. This program derives a new property for a triangle, the relation which holds among the lengths of the three edges and the surface area, from the three known properties.

```
:- public triangle/4.

surface_area(H,L,S) :- alg:L*H=2*S.
right(A,B,C) :- alg:A^2+B^2=C^2.
triangle(A,B,C,S) :-
        alg:C=CA+CB,
        right(CA,H,A),
        right(CB,H,B),
        surface_area(H,C,S).
```

The first clause, "surface_area", expresses the formula for computing the surface area S from the height H and the baseline length L. The second expresses the Pythagorean theorem for a right-angled triangle. The third asserts that every triangle can be divided into two right-angled triangles. (See Figure 1.).

In the following query, heron, shows the name of the file in which the CAL source program is defined.

```
?- alg:pre(s,10), heron:triangle(a,b,c,s).
```

This query asks for the general relationship between the lengths of the three edges and the surface area.

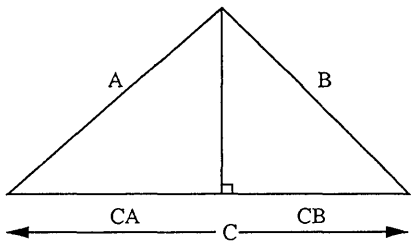


Figure 1: The third clause

The invocation of alg:pre(s,10) defines the precedence of the variable s to be 10. Since the algebraic constraint solver utilizes the Buchberger algorithm, ordering among monomials is essential for computation. This command changes the precedence of variables. Initially, the precedences of all variables are assigned to 0. Therefore, in this case, the precedence of variable s is raised.

To this query, the system responds with the following equation [1]:

```
s^2 = -1/16*b^4+1/8*a^2*b^2-1/16*a^4
      +1/8*c^2*b^2+1/8*c^2*a^2-1/16*c^4.
```

This equation is, actually, a developed form of Heron's formula.

When we call the query

```
?- heron:triangle(3,4,5,s).
```

the CAL system returns the following answer:

```
s^2 = 36
```

If a variable has finitely many values in all its solutions, there is a way of obtaining a univariate equation with the variable in the Gröbner base. Therefore, if we can add a function that enables us to compute the approximate values of the solutions of univariate equations, we can approximate all possible value of the variable.

For this purpose, we implemented a method of approximating the real roots of univariate polynomials. In CAL, all real roots of univariate polynomials are isolated by obtaining a set of intervals, each of which contains one real root. Then, each isolated real root is approximated by the given precision.

For application programs, we wanted to use approximate values to simplify other constraints. The general method to do this is to input equations of variables and their approximate values as constraints. For this purpose, we had to modify the original algorithm to compute Gröbner bases to accept approximate values.

When we call the query

[1] This equation represents the expression

$$s^2 = -\frac{1}{16}b^4 + \frac{1}{8}a^2b^2 - \frac{1}{16}a^4 + \frac{1}{8}c^2b^2 + \frac{1}{8}c^2a^2 - \frac{1}{16}c^4$$

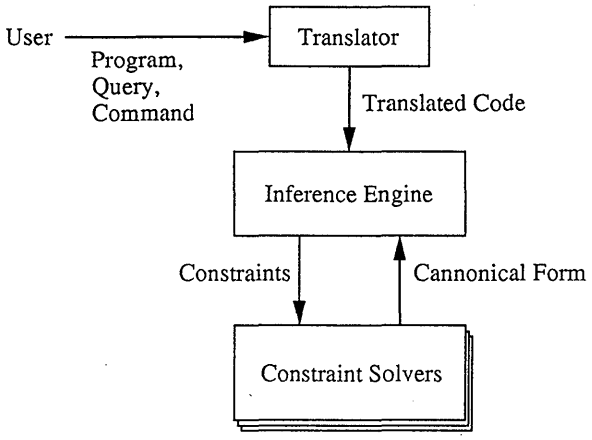


Figure 2: Overall construction of CAL language processor

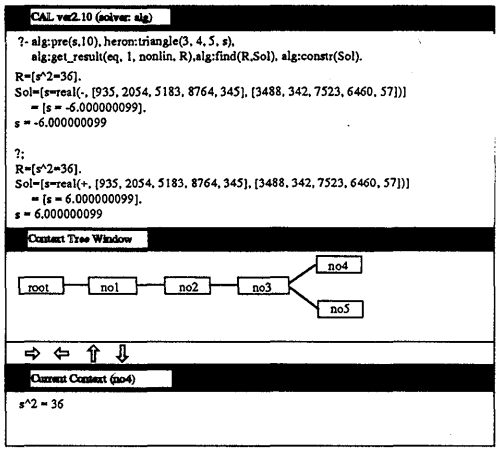


Figure 3: CAL system windows

```
?- alg:set_out_mode(float),
   alg:set_error1(1/1000000),
   alg:set_error2(1/100000000),
   heron:triangle(3,4,5,s),
   alg:get_result(eq,1,nonlin,R),
   alg:find(R,S),
   alg:constr(S).
```

we can obtain the answers `s = -6.000000099` and `s = 6.000000099`, successively by backtrack.

The first line of the above, `alg:set_out_mode`, sets the output mode to `float`. Without this, approximate values are output as fractions.

The second line of the above, `alg:set_error1`, specifies the precision used to compare coefficients in the computation of the Gröbner base. The third line, `set_error2`, specifies the precision used to approximate real roots by the bisection method.

The essence of the above query is invocations of `alg:get_result/4`, and `alg:find/2`. The fifth line, `alg:get_result`, selects appropriate equations from the Gröbner base. In this case, univariate (specified by `1`) non-linear (specified by `nonlin`) equations (specified by `eq`) are selected and unified to a variable `R`.

`R` is then passed to `alg:find` to approximate the real roots of equations in `R`. Such real roots are obtained in the variable `S`.

Then, `S` is again input as the constraint to reduce other constraints in the Gröbner base.

3.2 Configuration of CAL system

In this section, we will introduce the overall structure of the CAL system.

The CAL language processor consists of a translator, a inference engine, and constraint solvers. These subsystems are combined as shown in Figure 2.

The translator receives input from a user, and translates it into ESP code. Thus, a CAL source program is translated into the corresponding ESP program by the translator, which is executed by the inference engine. An appropriate constraint solver is invoked everytime the inference engine finds a constraint during execution.

The constraint solver adds the newly obtained constraint to the set of current constraints, and computes the canonical form of the new set.

At present, CAL offers the five constraint solvers discussed in Section 1.

3.3 Context

To deal with a situation in which a variable has more than one value, as in the above example, we introduced context and context tree.

A context is a set of constraints. A new context is created whenever the set is changed. In CAL, contexts are represented as nodes of a context tree. The root of a context tree is called the root context. The user is supposed to be in a certain context called the current context.

A context tree is changed in the following cases:

1. Goal execution:
   A new context is created as a child-node of the current context in the context tree.
2. Creation of a new set of constraints by requiring other answers for a goal:
   A new context is created as a sibling node of the current context in the context tree.
3. Changing the precedence:
   A new context is created as a child-node of the current context in the context tree.

In all cases, the newly created node represents the new set of constraints and becomes the current context.

Several commands are provided to manipulate the context tree: These include a command to display the contents of a context, a command to set a context as the

current context, and a command to delete the sub-tree of contexts from the context tree.

Figure 3 shows an example of the CAL processor window.

4 GDCC - Parallel CLP Programming Language

There are two major levels to parallelizing CLP systems. One is the execution of the Inference Engines and the Constraint Solvers in parallel. The other is the execution of a Constraint Solvers in parallel. There are several works on the parallelization of CLP systems: a proposal of ALPS [Maher 1987] introducing constraints into committed-choice language, a report of some preliminary experiments on integrating constraints into the PEPSys parallel logic system [Van Hentenryck 1989], and a framework of concurrent constraint (cc) language for integrating constraint programming with concurrent logic programming languages [Saraswat 1989].

The cc programming language paradigm models computation as the interaction among multiple cooperating agents through the exchange of query and assertion messages into a central store as shown in Figure 4.

In Figure 4, query information to the central store is represented as *Ask* and assertion information is represented as *Tell*.

This paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable), with respect to the current state of the store. Thus, this paradigm has high affinity with KL1 [Ueda and Chikayama 1990], our basic parallel language.

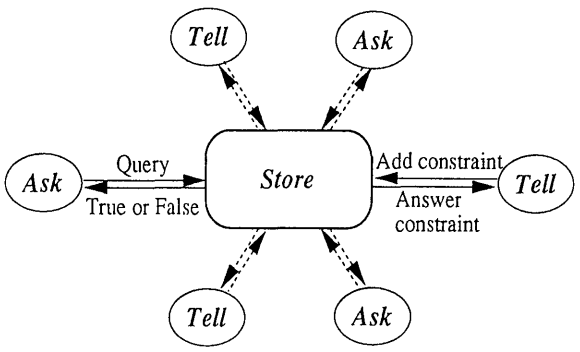


Figure 4: The cc language schema

GDCC (Guarded Definite Clauses with Constraints), which satisfies two level parallelism, is a parallel CLP language introducing the framework of cc. It is implemented in KL1 and is currently running on the Multi-PSI machine. GDCC includes most of KL1, since KL1 built-in predicates and unification can be regarded as a distinguished domain called HERBRAND [Saraswat 1989].

GDCC contains *Store*, a central database to save the canonical forms of constraints. Whenever the system meets an *Ask* or *Tell* constraint, the system sends it to the proper solver. *Ask* constraints are only allowed passive constraints which can be solved without changing the content of the *Store*. While in the *Tell* part, constraints which may change the *Store* can be written. In the GDCC program, only *Ask* constraints can be written in guards. This is similar to the KL1 guard in which active unification is inhibited.

GDCC supports multiple plug-in constraint solvers so that the user can easily specify a proper solver for a domain.

In this section, we briefly explain the language syntax of GDCC and its computation model. Then, the outline of the system is described. For further information about the implementation and the language specification, refer to [Terasaki *et al.* 1992].

4.1 GDCC language

A clause in GDCC has the following syntax:

Head :- *Ask* | *Tell*, *Goal*.

where, *Head* is a head part of a clause, "|" is a commit operator, *Goal* is a sequence of predicate invocations, *Ask* denotes Ask-constraints and invocations of KL1 built-in guard predicates, and *Tell* means Tell-constraints.

A clause is *entailed* if and only if *Ask* is reduced to *true*. Any clause with guards which cannot be reduced to either *true* or *false* is suspended. The body part, the right hand side of the commit operator, is evaluated if and only if *Ask* is *entailed*. Clauses whose guards are reduced true are called candidate clauses. A GDCC program fails when either all candidate clauses are rejected or there is a failure in evaluating *Tell* or *Goals*.

The next program is pony_and_man written in GDCC:

```
pony_and_man(Heads,Legs,Ponies,Men) :- true |
    alg# Heads= Ponies + Men,
    alg# Legs=  4*Ponies + 2*Men.
```

where, `true` is an *Ask* constraint which is always reduced as *true*. In the body, equations which begin with `alg#` are *Tell* constraints. `alg#` indicates that the constraints are solved by the algebraic solver. In a body part, not only *Tell* constraints but normal KL1 predicates can be written as well. Bi-directionality in evaluation of constraints, an important characteristic of CLP, is not spoiled by this limitation. For example, the query

```
?-  pony_and_man(5,14,Ponies,Men).
```

will return `Ponies=2`, and `Men=3`, and the query

```
?- pony_and_man(Heads,Legs,2,3).
```
will return Heads=5, and Legs=14, same as in CAL.

4.2 GDCC system

The GDCC system consists of the compiler, the shell, the interface and the constraint solvers. The compiler translates a GDCC source program into KL1 code. The shell translates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace and spy functions, together with solver-level event logging. The shell also provides limited support for incremental querying. The interface interacts with a GDCC program (object code), sends body constraints to a solver and checks guard constraints using results from a solver.

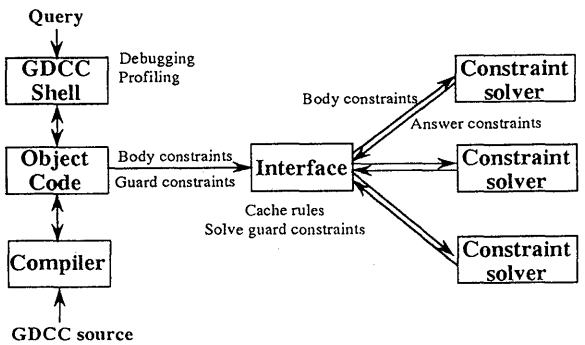


Figure 5: System Configuration of GDCC

The GDCC system is shown in Figure 5. The components are concurrent processes. Specifically, a GDCC program and the constraint solvers may execute in parallel, synchronizing only when, and to the extent, necessary at the program's guard constraints. That is, program execution proceeds by selecting a clause, and attempting to solve the guards of all its clauses in parallel. If one guard succeeds, the evaluation of the other guards is abandoned, and execution of the body can begin. In parallel with execution of the body goals by the inference engine, any constraints occurring in the body are passed to the constraint solver as they are being produced by the inference engine. This style of cooperation is very loosely synchronized and more declarative than sequential CLP.

4.3 Block

In order to apply GDCC to problems such as handling robot design problem [Sato and Aiba 1991], there were two major issues: handling multiple environments and synchronizing the inference engine with the constraint solvers. For instance, when the solution $X^2 = 2$ is derived from the algebraic solver, it must be solved in more detail using a function to compute the approximate real roots in univariate equations. There are two constraint sets in this example, one includes $X = \sqrt{2}$ and the other includes $X = -\sqrt{2}$. In the CAL system, the system selects one constraint set from these two and solves it, then, the other is computed by backtracking (i. e. , a system forces a failure). In committed-choice language GDCC, however, we cannot use backtracking to handle multiple environments. A similar problem occurs when a meta operation to constraint sets is required such as when computing a maximum value with respect to a given objective function. Before executing a meta operation, all target constraints must be sent to the solver. In a sequential CLP, this can be controlled when this description is written in a program. While in GDCC, we need another kind of mechanism to specify a synchronization point, since the sequence of clauses in a program does not relate to the execution sequence.

Introducing local constraint sets, however, which are independent to the global ones, can eliminate these problems. Multiple environments are realized by considering each multiple local constraint as one context. An inference engine and constraint solvers can be synchronized after evaluating a local constraint set.

Therefore, we introduced a mechanism called *block* to describe the scope of a constraint set. We can solve a certain goal sequence with respect to a local constraint set in a block. To encapsulate failure in a block, the *shoen* mechanism of PIMOS [Chikayama *et al.* 1988] is used.

5 Constraint Solvers and Parallelization

In this section, constraint solvers for both CAL and GDCC are briefly described. First, we describe the algebraic constraint solver for both CAL and GDCC. Then, we describe two Boolean constraint solvers – one is a solver utilizing the modified Buchberger algorithm and the other is a solver utilizing the incremental Boolean elimination algorithm. The former is for both CAL and GDCC, while the later is for CAL alone. Third, an integer constraint solver for GDCC is described, and fourth, a hierarchical constraint solver for CAL and GDCC is described. In the next subsection,a set constraint solver for CAL is described. And in the last subsection, a preliminary consideration on efficiency improvement of the algebraic constraint solver by applying dependency analysis of constraints.

All constraint solvers for CAL are written in ESP, and those for GDCC are written in KL1.

5.1 Algebraic Constraint Solver

The constraint domain of the algebraic solver is multivariate (non-linear) algebraic equations. The Buchberger

algorithm [Buchberger 1985] is a method to solve nonlinear algebraic equations which have been widely used in computer algebra over the past years.

Recently, several attempts have been made to parallelize the Buchberger algorithm, with generally disappointing results in absolute performance [Ponder 1990, Senechaud 1990, Siegl 1990], except in shared-memory machines [Vidal 1990, Clarke *et al.* 1990]. We parallelize the Buchberger algorithm while laying emphasis on absolute performance and incrementality rather than on deceptive parallel speedup. We have implemented several versions and continue to improve the algorithm.

In this section, we outline both the sequential version and the parallel version of the Buchberger algorithm.

5.1.1 Gröbner base and Buchberger algorithm

Without loss of generality, we can assume that all polynomial equations are in the form of $p = 0$. Let $E = \{p_1 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations. Buchberger introduced the notion of a Gröbner base and devised an algorithm to compute the basis of a given set of polynomials. A rough sketch of the algorithm is as follows (see [Buchberger 1985] for a precise definition).

Let a certain ordering among monomials and a system of polynomials be given. An equation can be considered a rewrite rule which rewrites the greatest monomial in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is $Z > X > B > A$, a polynomial equation, $Z - X + B = A$, can be considered to be the rewrite rule, $Z \rightarrow X - B + A$. A pair of rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, is called a *critical pair*, since the least common multiple of their left-hand sides can be rewritten in two different ways. The S-polynomial of such a pair is defined as:

$$\text{S-poly}(L_1, L_2) = R_1 \frac{lcm(L_1, L_2)}{L_2} - R_2 \frac{lcm(L_1, L_2)}{L_1}$$

where $lcm(L_1, L_2)$ represents the least common multiplier of L_1 and L_2.

If further rewriting does not succeed in rewriting the S-polynomial of a critical pair to zero, the pair is said to be *divergent* and the S-polynomial is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called a *Gröbner base* of the original system of equations.

If a Gröbner base does not have two rules, one of which rewrites the other, the Gröbner base is called *reduced*. The *reduced* Gröbner base can be considered a canonical form of the given constraint set since it is unique with respect to the given ordering of monomials. If all the solutions of a equation $f = 0$ are included in the solution set of E, then f is rewritten to zero by the Gröbner base of E. On the contrary, if a set of polynomials E has no solution, then the Gröbner base of E includes "1". Therefore, this algorithm has good properties for deciding the satisfiability of a given constraint set.

5.1.2 Parallel Algorithm

The coarse-grained parallelism in the Buchberger algorithm, suitable for the distributed memory machine, is the parallel rewriting of a set of polynomials. However, since the convergence rate of the Buchberger algorithm is very sensitive to the order in which polynomials are converted into rules, implementation must carefully select small polynomials at an early stage. We have implemented solvers in three different architectures; namely, a pipeline, a distributed architecture, and a master-slave architecture. We briefly mention here the master-slave architecture since this solver has comparatively good performance.

Figure 6 shows the architecture.

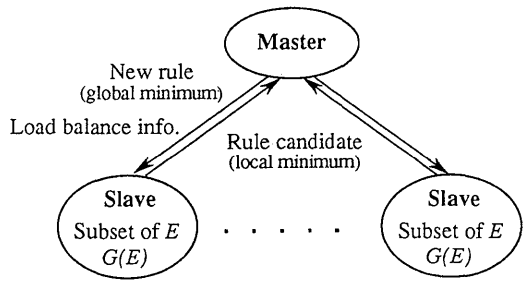


Figure 6: Architecture of master-slave type solver

The set of polynomials E is physically partitioned with each slave taking a different part. The initial rule set of $G(E)$ is duplicated so that all slaves use the same rule set. New polynomials are distributed to the slaves by the master. The outline of the reduction cycle is as follows.

Each slave rewrites its own polynomials by the $G(E)$, selects the local minimum polynomial from them, and sends its leading power product to the master. The master processor waits for reports from all the slaves, and selects the global minimum power products. The minimum polynomial can be decided only after all slaves finish reporting to the master. A polynomial, however, which is not the minimum can be decided quickly. Thus, the *not-minimum* message is sent to slaves as soon as possible, and the processors that receive the *not-minimum* message reduce polynomials by the old rule set while waiting for a new rule. While the slave is receiving the *minimum* message, the slave converts the polynomial into a new rule and sends it to the master. The master sends the new rule to all slaves except the owner. If more than one candidate have equal power products, then all of these

candidates are converted to rules by slaves and they go to final selection at the master.

Table 1 shows the results of the benchmark problems. The problems are adopted from [Boege *et al.* 1986, Backelin and Fröberg 1991]. Refer to [Terasaki *et al.* 1992] for further details.

Table 1: Timing and speedup of the master-slave arch.(unit:sec)

Problems	Processors				
	1	2	4	8	16
Katsura-4	8.90	7.00	5.83	6.53	9.26
	1	1.27	1.53	1.36	0.96
Katsura-5	86.74	57.81	39.88	31.89	36.00
	1	1.50	2.18	2.72	2.41
Cyc.5-roots	27.58	21.08	19.27	19.16	25.20
	1	1.31	1.43	1.44	1.10
Cyc.6-roots	1430.18	863.62	433.73	333.25	323.38
	1	1.66	3.30	4.29	4.42

5.2 Boolean Constraint Solver

There are several algorithms that solve Boolean constraints, but we do not know so many that we can get the canonical form of constraints, one that can calculate solutions incrementally and that uses no parameter variables. These criteria are important for using the algorithm as a constraint solver, as we described in Section 2. First, we implemented the Boolean Buchberger algorithm [Sato and Sakai 1988] for the CAL system, then we tried to parallelize it for the GDCC system. This algorithm satisfies all of these criteria. Moreover, we developed another sequential algorithm named Incremental Boolean elimination, that also satisfies all these criteria, and we implemented it for the CAL system.

5.2.1 Constraint Solver by Buchberger Algorithm

We first developed a Boolean constraint solver based on the modified Buchberger algorithm called the Boolean Buchberger algorithm [Sato and Sakai 1988, Aiba *et al.* 1988]. Unlike the Buchberger algorithm, it works on the Boolean ring instead of on the field of complex numbers. It calculates the canonical form of Boolean constraints called the Boolean Gröbner base. The constraint solver first transforms formulas including some Boolean operators such as *inclusive-or* ($\vee$) and/or *not* ($\neg$) to expressions on the Boolean ring before applying the algorithm.

We parallelized the Boolean Buchberger algorithm in KL1. First we analyzed the execution of the Boolean Buchberger algorithm on CAL for some examples, then we found the large parts that may be worth parallelizing, rewriting formulas by applying rules. We also tried to find parts in the algorithm which can be parallelized by analyzing the algorithm itself. Then, we decided to adopt a master-slave parallel execution model.

In a master-slave model, one master processor plays the role of the controller and the other slave processors become the reducers. The controller manages Boolean equations, updates the temporary Gröbner bases (**GB**) stored in all slaves, makes S-polynomials and self-critical pair polynomials, and distributes equations to the reducers. Each reducer has a copy of **GB** and reduces equations which come from the controller by **GB**, and returns non-zero reduced equations to the controller. When the controller becomes idle after distributing equations, the controller plays the role of a reducer during the process of reduction.

For the 6-queens problem, the speedup ratio of 16 processors to a single processor is 2.96. Because the parallel execution part of the problem is 77.7% of whole execution, the maximum speedup ratio is 4.48 in our model. The difference is due to the task distribution overhead, the update of **GB** in each reducer, and the imbalance of distributed tasks.

Then, we improved our implementation so as not to make redundant critical pairs. This improvement causes the ratio of parallel executable parts to decrease, so the improved version becomes faster than the original version, but the speedup ratio of 16 processors to a single processor drop to 2.28.

For more details on the parallel algorithm and results, refer to [Terasaki *et al.* 1992].

5.2.2 Constraint Solver by Incremental Boolean Elimination Algorithm

Boolean unification and SL-resolution are well known as Boolean constraint solving algorithms other than the Boolean Buchberger algorithm. Boolean unification is used in CHIP [Dincbas *et al.* 1988] and SL-resolution is used in Prolog III [Colmerauer 1987]. Boolean unification itself is an efficient method. It becomes even more efficient using the binary decision diagrams (BDD) as data structures to represent Boolean formulas. Because the solutions by Boolean unification include extra variables introduced during execution, it cannot calculate any canonical form of the given constraints if we execute it incrementally. For this reason, we developed a new algorithm, *Incremental Boolean elimination*. As with the Boolean unification, this algorithm is based on Boole's elimination, but it introduces no extra variables, and it can calculate a canonical form of the given Boolean constraints.

We denote Boolean variables by $x, y, z, \ldots$, and Boolean polynomials by $A, B, C, \ldots$. We represent all Boolean formulas only by logical connectives *and* ($\times$) and *exclusive-or* ($+$). For example, we can represent Boolean formulas $F \wedge G$, $F \vee G$ and $\neg F$ by $F \times G$, $F \times G + F + G$ and $F + 1$. We use the expression $F_{x=G}$ to represent the formula obtained by substituting all occurrences of variable x in formula F with formula G. We omit $\times$ symbols

as usual when there is no confusion. We assume that there is a total order over variables.

We define the *normal* Boolean polynomials recursively as follows.

1. The two constants 0, and 1 are *normal.*
2. If two normal Boolean polynomials A and B consist of only variables smaller than x, then $Ax + B$ is *normal*, and we denote it by $Ax \oplus B$. We call A the *coefficient of* x.

If variable x is at a maximum in formula F, then we can transform F to the normal formula $(F_{x=0}+F_{x=1})x \oplus F_{x=0}$. Hence we assume that all polynomials are normal.

Boole's elimination says that if a Boolean formula F is 0, then $F_{x=0} \times F_{x=1}$ $(= G)$ is also 0. Because G does not include x, if F includes x, then G includes fewer variables than F. Similarly we can get polynomials with fewer variables gradually by Boole's eliminations.

Boolean unification unifies x with $(F_{x=0}+F_{x=1}+1)u+F_{x=0}$ after eliminating variable x from formula F, where u is a free extra variable. This unification means the substitution x with $(F_{x=0}+F_{x=1}+1)u+F_{x=0}$, when a new Boolean constraint with variable x is given, the result of the substitution contains u instead of x. Therefore, Boolean unification unifies u with a formula with another extra variable.

Incremental Boolean elimination applies the following reduction to every formula instead of transforming $F = 0$ to $x = (F_{x=0} + F_{x=1} + 1)u + F_{x=0}$ and unifying x with $(F_{x=0} + F_{x=1} + 1)u + F_{x=0}$. That is why the Incremental Boolean elimination needs no extra variables.

Reduction A formula Cx $(C \not\equiv 1)$ is reduced by the formula $Ax \oplus B = 0$ shown below. This reduction tries to reduce the coefficient of x to 1 if possible, otherwise it tries to reduce it to the smallest formula possible.

$$\begin{array}{ll} Cx \rightarrow x + BC + B & (AC + A + C \equiv 1) \\ Cx \rightarrow (A + 1)Cx + BC & (\text{otherwise}) \end{array}$$

When a new Boolean constraint is given, the following operation is executed, since Incremental Boolean elimination does not execute unification.

Merge Operation Let $Cx \oplus D = 0$ be a new constraint, and suppose that we have a constraint $Ax \oplus B = 0$. Then we make the merged constraint $(AC+A+C)x \oplus (BD + B + D) = 0$ the new solution. If the normal form of $ACD + BC + CD + D$ is not 0, we successively apply the merge operation to it.

This operation is an expansion of Boole's elimination. That is, if we have no constraint yet, we can consider A and B as 0. In this case, the merge operation is the same as Boole's elimination.

Example Consider the following constraints. Exactly one of five variables a, b, c, d, e $(a < b < c < d < e)$ is 1.

$$a \wedge b = 0,\ a \wedge c = 0,\ a \wedge d = 0,\ a \wedge e = 0,\ b \wedge c = 0,\ b \wedge d = 0,\ b \wedge e = 0,\ c \wedge d = 0,\ c \wedge e = 0,\ d \wedge e = 0,\ a \vee b \vee c \vee d \vee e = 1$$

By Incremental Boolean elimination, we can obtain the following canonical solution.

$$\begin{array}{rcl} e & = & d+c+b+a+1 \\ (c+b+a) \times d & = & 0 \\ (b+a) \times c & = & 0 \\ a \times b & = & 0 \end{array}$$

The solution can be interpreted as follows. Because the solution does not have an equation of the form $A \times a = B$, variable a is free. Because $a \times b = 0$, if $a = 1$ then the variable b is 0. Otherwise b is free. The discussion continues and, finally, because $e = d+c+b+a+1$, if a, b, c, d are all 0, then variable e is 1. Otherwise e is 0.

By assignment of 0 or 1 to all variables in increasing order of $<$ under a solution by Boolean Incremental elimination, we can easily obtain any assignments that satisfy the given constraints. Thus, by introducing an adequate order to variables, we can obtain a favorite enumeration of assignments satisfy the given constraints.

5.3 Integer Linear Constraint Solver

The constraint solver for the integer linear domain checks the consistency of the given equalities and inequalities of the rational coefficients, and, furthermore, gives the maximum or minimum values of the objective linear function under these constraint conditions. The purpose of this constraint solver is to provide an efficient constraint solver for the integer optimization domain by achieving a computation speedup incorporating parallel execution into the search process.

The integer linear solver utilizes the rational linear solver (parallel linear constraint solver) for the optimization procedure to obtain an evaluation of relaxed linear problems created in the course of its solution. A rational linear solver is realized by the simplex algorithm. We implemented the integer linear constraint solver for GDCC.

5.3.1 Integer Linear Programming and Branch and Bound Method

In the following, we discuss a parallel search method employed in this integer linear constraint solver. The problem we are addressing is a mixed integer programming problem, namely, to find the maximum or minimum value of a given linear function under the integer linear constraints.

The problem can be defined as follows: The problem is to minimize the following objective function on variables

x_j which run on real numbers, and variables y_j which run on integers:

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_i + \sum_{i=1}^{m} b_{ij} y_i \geq e_j, \text{ for } j = 1, \ldots, l,$$

$$\sum_{i=1}^{n} c_{ij} x_i + \sum_{i=1}^{m} d_{ij} y_i = f_j, \text{ for } j = 1, \ldots, k,$$

where

$$x_i \in R, \text{ and } x_i \geq 0, \text{ for } i = 1, \ldots, n$$

$$y_i \in Z, \text{ where } l_i \leq y_i \leq u_i,$$

$$l_j, u_j \in Z, \text{ for } i = 1, \ldots, m$$

and

$$a_{ij},\ b_{ij},\ c_{ij},\ d_{ij},\ e_i,\ f_i \text{ are real constants.}$$

The method we use is the Branch-and-Bound algorithm. Our algorithm checks in the first place the solution of the original problem without requiring variables y_i in the above to take integer value. We call this problem a continuously relaxed problem. If the continuously relaxed problem does not have an integer solution, then we proceed by dividing the original problem into two sub-problems successively, producing a tree structured search space.

Continuously relaxed problems can be solved by the simplex algorithm, and if the original integer variables have exact integer values, then it yields the solution to the integer problem. Otherwise, we select an integer variable y_s which takes a non-integer value $\bar{y}_s$ for the solution of continuously relaxed problems, and imposes two different interval constraints derived from neighboring integers of the value $\bar{y}_s$, $l_s \leq y_s \leq [\bar{y}_s]$ and $[\bar{y}_s] + 1 \leq y_s \leq u_s$ to the already existing constraints, and obtains two child problems (See Figure 7). Continuing this procedure, which is called branching, we go on dividing the search space to produce more constrained sub-problems. Eventually this process leads to a sub-problem with the continuous solution which is also the integer solution of the problem. We can select the best integer solution from among those found in the process.

While the above branching process only enumerates integer solutions, if we have a measure to guarantee that a sub-problem cannot have a better solution compared to the already obtained integer solution in terms of the optimum value of the objective function, then we can skip that sub-problem and only need to search the rest of the nodes. Continuously relaxed problems give a measure for this, since these relaxed problems always have better optimum values for the objective function than the original integer problems. Sub-problems whose continuously relaxed problems have no better optimum than the integer

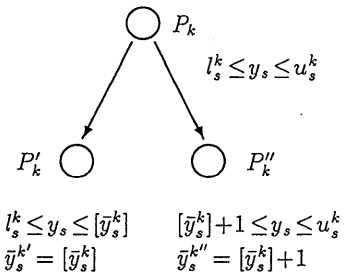


Figure 7: Branching of Nodes

solution obtained already cannot give a better optimum value, which means it is unnecessary to search further (bounding procedure).

We call these sub-problems obtained through the branching process search nodes.

The following two important factors decide the order in which the sequential search process goes through nodes in the search space:

1. The priorities of sub-problems(nodes) in deciding the next node on which the branching process works.

2. Selection of a variable out of the integer variables with which the search space is divided.

It is preferable that the above selections are done in such a way that the actual nodes searched in the process of finding the optimal form as small a part of the total search space as possible. We adopted one of the best heuristics of this type from operations research as a basis of our parallel algorithm([Benichou *et al.* 1971]).

5.3.2 Parallelization of Branch-and-Bound Method

As a parallelization of the Branch-and-Bound algorithm, we distribute search nodes created through the branching process to different processors, and let these processors work on their own sub-problems following a sequential search algorithm. Each sequential search process communicates with other processes to transmit information on the most recently found solutions and on pruning sub-nodes, thus making the search proceed over a network of processors. We adopted one of the best search heuristics used in sequential algorithms. Heuristics are used for controlling the schedule of the order of sub-nodes to be searched, in order to reduce the number of nodes needed to get to the final result. Therefore, it is important in designing parallel versions of search algorithms to balance the distributed load among processors, and to communicate information for pruning as fast as possible between these processors.

We considered a parallel algorithm design derived from the above sequential algorithm to be implemented on the distributed memory parallel machine Multi-PSI.

Our parallel algorithm exploits the independence of many sub-processes created through the branching procedure in the sequential algorithm and distributes these processes to different processors (see Figure 8). Scheduling of sub-problems is done by the use of the priority control facility provided from the KL1 language (See[Oki *et al.* 1989]). The incumbent solutions are transferred between processors as global data to be shared so that each processor can update the current incumbent solution as soon as possible.

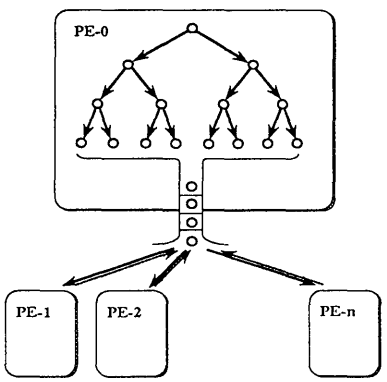


Figure 8: Generation of Parallel Processes

5.3.3 Experimental Results

We implemented the above parallel algorithm in the KL1 language and experimented with the job-shop scheduling problem as an example of mixed-integer problems. Below are the results of computation speedups for a "4 job 3 machine" problem and the total number of searched nodes to get to the solution.

Table 2: Speedup of the Integer Linear Constraint Solver

processors	1	2	4	8
speedup	1.0	1.5	1.9	2.3
number of nodes	242	248	395	490

The above table shows the increase of the number of searched nodes as the number of processors grows. This is for one reason because of the speculative computation inherent in this type of parallel algorithm. Another reason is that the communication latency produces unnecessary computation which could have been avoided if incumbent solutions are communicated instantaneously from the other processor and the unnecessary nodes are pruned.

It is in this way that we get the problem in parallel programming of how to reduce the growth in size of the total search space when multi-processors are used compared with that traversed on one processor using sequential algorithms.

5.4 Hierarchical Constraint Solver

5.4.1 Soft Constraints and Constraint Hierarchies

We have proposed a logical foundation of soft constraints in [Satoh 1990] by using a meta-language which expresses interpretation ordering. The idea of formalizing soft constraints is as follows. Let hard constraints be represented in first-order formulas. Then an interpretation which satisfies all of these first-order formulas can be regarded as a possible solution and soft constraints can be regarded as an order over those interpretations because soft constraints represent criteria applying to possible solutions for choosing the most preferred solutions. We use a meta-language which represents a preference order directly. This meta-language can be translated into a second-order formula to provide a syntactical definition of the most preferred solutions.

Although this framework is rigorous and declarative, it is not computable in general because it is defined by a second-order formula. Therefore, we have to restrict the class of constraints so that these constraints are computable.

Therefore, we introduce the following restriction to make the framework computable.

1. We fix the considered domain so that interpretations of domain-dependent relations are fixed.
2. Soft and hard constraints consist of domain-dependent relations only.

If we accept this restriction, the soft constraints can be expressed in a first-order formula. Moreover, there is a relationship between the above restricted class of soft constraints and hierarchical CLP languages (HCLP languages) [Borning *et al.* 1989, Satoh and Aiba 1990b], as shown in [Satoh and Aiba 1990a].

HCLP language is a language augmenting CLP language with labeled constraints. An HCLP program consists of rules of the form:

$$h \text{ :- } b_1,...,b_n$$

where h is a predicate, and $b_1,...,b_n$ are predicate invocations or constraints or labeled constraints. Labeled constraints are of the form:

$$\texttt{label}\ C$$

where C is a constraint in which only domain-dependent functional symbols can be functional symbols and `label` is a label which expresses the strength of the constraint C.

As shown in [Satoh and Aiba 1990a], we can calculate the most preferable solutions by constraint hierarchies

in the HCLP language. Based on this correspondence, we have implemented an algorithm for solving constraint hierarchy on the PSI machine with the following features.

1. There are no redundant calls of the constraint solver for the same combination of constraints since it calculates reduced constraints in a bottom-up manner.
2. If an inconsistent combination of constraints is found by calling the constraint solver, it is registered as a nogood and is used for detecting further contradiction. Any extension of the combination will not be processed so as to avoid unnecessary combinations.
3. Inconsistency is detected without a call of the constraint solver if a processed combination subsumes a registered nogood.

In [Borning *et al.* 1989], Borning et al. give an algorithm for the solving constraint hierarchy. However, it uses backtracking to get an alternative solution and so may redundantly call the constraint solver for the same combination of constraints.

Our implemented language is called CHAL (Contrainte Hierarchiques avec Logique) [Satoh and Aiba 1990b], and is an extension of CAL.

5.4.2 Parallel Solver for Constraint Hierarchies

The algorithms we have implemented on the PSI machine have the following parallelism.

1. Since we construct a consistent constraint set in a bottom-up manner, the check for consistency for each independent constraint set can be done in parallel.
2. We can check if a constraint set is included in nogoods in parallel for each independent constraint set.
3. There is parallelism inside a domain-dependent constraint solver.
4. We can check for answer redundancy in parallel.

Among these parallelisms, the first one is the most coarse and the most suitable for implementation on the Multi-PSI machine. So, we exploit the first parallelism. Then, features of the parallel algorithm become the following.

1. Each processor constructs a maximal consistent constraint set from a given constraint set in a bottom-up manner in parallel. However, once a constraint set is given, there is no distribution of tasks. So, we make idle processors require some task from busy processors and if a busy processor can divide its task, then it sends the task to the idle processor.
2. By pre-evaluation of a parallel algorithm, we found that the nogood subsumption check and the redundancy check have very large overheads. So, we do not check nogood subsumptions and we check redundancy only at the last stage of execution.

Table 3: Performance of Parallel Hierarchical Constraint Solver(unit: sec)

	Processors				
problems	1	2	4	8	16
Tele4	43	32	32	32	29
	1	1.34	1.34	1.34	1.48
5queen	69	39	26	21	19
	1	1.77	2.65	3.29	3.63
6queen	517	264	136	77	50
	1	1.96	3.80	6.71	10.34

Table 3 shows the speedup ration for three examples. Tele4 is to solve ambiguity in natural language phrases. 5queen and 6queen are to solve the 5 queens and 6 queens problem. We represent these problems in Boolean constraints and use the Boolean Buchberger algorithm [Sato and Sakai 1988, Sakai and Aiba 1989] to solve the constraints.

According to Table 3, we obtain 1.34 speedup for Tele4, 3.63 speedup for 5queen, and 10.34 speedup for 6queen. Although 6queen is a large problem for the Boolean Buchberger algorithm and gives us the largest speedup, the speedup saturates at around 16 processors. This expresses that the load is not well-distributed and we have to look for a better load-balancing method in the future.

5.5 Set Constraint Solver

The set constraint solver handles any kind of constraint presented in the following conjunction of predicates.

$$\begin{array}{c} F_1(\bar{x}, \bar{X}) \\ \vdots \\ F_n(\bar{x}, \bar{X}) \end{array}$$

where each predicate $F_i(\bar{x}, \bar{X})$ is a predicate constructed from predicate symbols $\in$, $\subseteq$, $\neq$ and $=$, function symbols $\cap$, $\cup$, and $\cdot^{\sim}$, element variables $\bar{x}$, and set variables $\bar{X}$, and some symbols of constant elements.

For the above constraints, the solver gives the answer of the form:

$$\begin{array}{c} f_1(\bar{x}, \bar{X}) = 0 \\ \vdots \\ f_l(\bar{x}, \bar{X}) = 0 \\ h_1(\bar{x}) = 0 \\ \vdots \\ h_m(\bar{x}) = 0 \end{array}$$

where $h_1(\bar{x}) = 0, \ldots, h_m(\bar{x}) = 0$ give the necessary and sufficient conditions for satisfying the constraints. Moreover, for each solution for the element variables, the system of whole equations instantiated by the solution puts the original constraints into a normal form (i.e. a solution).

For more detailed information on the constraint solver, refer to [Sato *et al.* 1991].

Let us first consider the following example.

$$\begin{aligned} A^{\sim} \cap C^{\sim} \cap E^{\sim} &= \emptyset \\ C \cup E &\supseteq B \\ C \cup E &\supseteq D \\ D \cap B^{\sim} &\supseteq A \\ A^{\sim} \cap B &\subseteq D \\ A \cup B &\supseteq D \end{aligned}$$

where the notation $A^{\sim}$ denotes the complement of A.

Since a class of sets forms a Boolean algebra, this constraint can be considered a Boolean constraint. Hence we can solve this by computing its Boolean Gröbner base:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \end{aligned}$$

We should note that there is neither an element variable nor a constant on elements in the above constraints. Hence they can be expressed as Boolean equations with variables A, B, C, D and E. This, however, does not necessarily hold in every constraint of sets.

Consider the following constraints with an additional three predicates including elements.

$$\begin{aligned} A^{\sim} \cap C^{\sim} \cap E^{\sim} &= \emptyset \\ C \cup E &\supseteq B \\ C \cup E &\supseteq D \\ D \cap B^{\sim} &\supseteq A \\ A^{\sim} \cap B &\subseteq D \\ A \cup B &\supseteq D \\ (C \cap \{x\}) \cup (E \cap \{p\}) &= D \cap \{x, p\} \\ x &\notin A \\ p &\notin B \end{aligned}$$

where x is an element variable and p is a constant symbol of an element.

This can no longer be represented with the Boolean equations as above. For example the last formula is expressed as $\{p\} * B = 0$, where $\{p\}$ is considered a coefficient. In order to handle such general Boolean equations, we extended the notion of Boolean Gröbner bases [Sato *et al.* 1991], which enabled us to implement the set constraint solver.

For the above constraint, the solver gives the following answer:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \\ \{x\} * E * B &= \{x\} * E + \{x\} * B + \{x\} \\ \{p\} * C * A &= \{p\} * C + \{p\} * A + \{p\} \\ \{p\} * E &= \{p\} * A \\ \{x\} * C &= \{x\} * B \\ \{x\} * A &= 0 \\ \{p\} * B &= 0 \\ \{p\} * \{x\} &= 0 \end{aligned}$$

In this example, $\{p\} * \{x\} = 0$ is the satisfiability condition. This holds if and only if $x \neq p$. In this case, there are always A, B, C and D that satisfy the original constraints. The normal form is:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \\ \{x\} * E * B &= \{x\} * E + \{x\} * B + \{x\} \\ \{p\} * C * A &= \{p\} * C + \{p\} * A + \{p\} \\ \{p\} * E &= \{p\} * A \\ \{x\} * C &= \{x\} * B \\ \{x\} * A &= 0 \\ \{p\} * B &= 0 \end{aligned}$$

5.6 Dependency Analysis of Constraint Set

From several experiments on writing application programs, we can conclude that the powerful expressiveness of these languages is a great aid to programming, since all users have to do to describe a program is to define the essential properties of the problem itself. That is, there is no need to describe a method to solve the problem.

On the other hand, sometimes the generality and power of constraint solvers turn out to be a drawback for these languages. That is, in some cases, especially for very powerful constraint solvers like the algebraic constraint solver in CAL or GDCC, it is difficult to implement them efficiently because of their generalities, in spite of great efforts.

As a subsystem of language processors, efficiency in constraint solving is, of course, one of the major issues in the implementation of those language processors [Marriott and Sondergaard 1990, Cohen 1990].

In general, for a certain constraint set, the efficiency of constraint solving is strongly dependent on the order in which constraints are input to a constraint solver. However, in sequential CLP languages like CAL, this order is determined by the position of constraints in a program, because a constraint solver solves constraints accumulated by the inference engine that follows SLD-resolution.

In parallel CLP languages like GDCC, the order of constraints input to a constraint solver is more important than in sequential languages. Since an inference engine and constraint solvers can run in parallel, the order of constraints is not determined by their position in a program. Therefore, the execution time may vary according to the order of constraints input to the constraint solver.

In CAL and GDCC, the computation of a Gröbner base is time-consuming and it is well known that the Buchberger algorithm is doubly exponential in worst-case complexity [Hofmann 1989]. Therefore, it is worthwhile to rearrange the order of constraints to make the constraint solver efficient.

We actually started research into the order of constraints based on dependency analysis [Nagai 1991, Nagai and Hasegawa 1991]. This analysis consisted of dataflow analysis, constraint set collection, dependency analysis on constraint sets, and determination of the ordering of goals and the preference of variables.

To analyze dataflow, we use top-down analysis based on SLD-refutation. For a given goal and a program, the invocation of predicates starts from the goal without invoking a constraint solver, and variable bindings and constraints are collected.

In this analysis, constraints are described in terms of graphical (bipartite graph) representation. An algebraic structure of s set of constraints is extracted using DM decomposition [Dulmage and Mendelsohn 1963], which computes a block upper triangular matrix by canonical reordering a matrix corresponding to the set of constraints.

As a result of analysis, a set of constraints can be partitioned into relatively independent subsets of constraints. These partitions are obtained so that the number of variables shared among different blocks is as small as possible. Besides this partition, shared variables among partitions and shared variables among constraints inside of a block are also obtained. Based on these results, the order of goals and the precedence of variables are determined.

We show the results of this method for two geometric theorem proving problems [Kapur and Mundy 1988, Kutzler 1988]: one is the theorem that three perpendicular bisectors of three edges of a triangle intersect at a point, and the other is the, so-called, nine points circle theorem. The former theorem can be represented by 5 constraints with 8 variables and gives about 3.2 times improvement. The latter theorem can be represented by 9 constraints with 12 variables and gives about 276 times improvement.

6 CAL and GDCC Application Systems

To show the feasibility of CAL and GDCC, we implemented several application systems. In this section, two of these, the handling robot design support system and the Voronoi diagram construction program, are described.

6.1 Handling Robot Design Support System

The design process of a handling robot consists of a fundamental structure design and a internal structure design [Takano 1986]. The fundamental structure design determines the framework of the robot, such as the degree of freedom, number of joints, and arm length. The internal structure design determines the internal details of the robot, such as the mortar torque of each joint. The handling robot design support system mainly supports the fundamental structure design.

Currently, the method to design a handling robot is as follows:

1. First, the type of the robot, such as cartesian manipulator, cylindlical manipulator, or articulated manipulator has to be decided according to the requirements for the robot.
2. Then, a system of equations representing the relation between the end effector and joints is deduced. Then the system of equations is transformed to obtain the desired form of equations.
3. Next, a program to analyze the robot being designed is coded by using an imperative programming language, such as Fortran or C.
4. By executing the program, the design is evaluated. If the result is satisfactory, then the design process terminates, otherwise, the whole process should be repeated until the result satisfies the requirements.

By adopting the CLP paradigm to the design process of a handling robot, through coding a CLP program representing the relation obtained in 2 in the above, the transformation can be done by executing the program. Thus, processes 2 and 3 can be supported by a computer.

6.1.1 Kinematics and Statics Program by Constraint Programming

Robot kinematics represents the relation between a position and the orientation of the end effector, the length of each arm, and the rotation angle of each joint. We call a position and an orientation of the end effector, hand parameters, and we call the rest, joint parameters. Robot statics represent the relation between joint parameters: force working on the end-effector, and torque working on each joint [Tohyama 1989]. These relations are essential for analyzing and evaluating the structure of a handling robot.

To make a program that handles handling robot structures, we have to describe a program independent of its fundamental structure. That is, kinematics and statics programs are constructed to handle any structure of robot by simply changing a query.

Actually, these programs receive a matrix which represents the structure of a handling robot being designed in terms of a list of lists. By manipulating the structure of this argument, any type of handling robot can be handled by the one program.

For example, the following query asks the kinematics of a handling robot with three joints and three arms.

```
robot([[cos3, sin3, 0,  0, z3, 0, 0, 1],
       [cos2, sin2, x2, 0, 0,  1, 0, 0],
       [cos1, sin1, 0,  0, z1, 0, 0, 1]],
```

```
5, 0, 0, 1, 0, 0, 0, 1, 0,
px, py, pz, ax, ay, az, cx, cy, cz).
```

where the first argument represents the structure of the handling robot, px, py, and pz represents a position, ax, ay, az, cx, cy, and cz represents an orientation by defining two unit vectors which are perpendicular to each other. sin's and cos's represent the rotation angle of each joint, and z3, x2, and z1 represent the length of each arm. For this query, the program returns the following answer.

```
cos1^2 = 1-sin1^2
cos2^2 = 1-sin2^2
cos3^2 = 1-sin3^2
    px = -5*cos2*sin3*sin1+z_3*sin2*sin1
         +5*cos3*cos1+x_2*cos1
    py = 5*cos3*sin1+x_2*sin1
         +5*cos1*cos2*sin3-z_3*cos1*sin2
    pz = 5*sin3*sin2+z_1+z_3*cos2
    ax = -1*cos2*sin3*sin1+cos3*cos1
    ay = cos3*sin1+cos1*cos2*sin3
    az = sin3*sin2
    cx = -1*cos1*sin3-cos3*cos2*sin1
    cy = -1*sin3*sin1+cos3*cos1*cos2
    cz = cos3*sin2
```

That is, the parameters of the position and the orientation are expressed in terms of the length of each arm and the rotation angle of each joint.

Note that this kinematics program has the full features of the CLP program. The problem of calculating hand parameters from joint parameters is called forward kinematics, and the converse is called inverse kinematics. We can deal with both of them with the same program.

This program can be seen as a generator of programs dealing with any handling robot which has a user designed fundamental structure.

Statics has the same features as the kinematics program described. That is, the program can deal with any type of handling robot by simply changing its query.

6.1.2 Construction of Design Support System

The handling robot design support system should have the following functions

1. to generate the constraint representing kinematics and statics for any type of robot,
2. to solve forward and inverse kinematics,
3. to calculate the torque which works on each joint, and
4. to evaluate manipulability.

The handling robot design support system consists of the following three GDCC programs in order to realize these functions,

Kinematics a kinematics program
Statics a statics program
Determinant a program to calculate the determinant of a matrix

Kinematics and **Statics** are the programs we described above. A matrix to evaluate the manipulability of a handling robot, called a Jacobian matrix, is obtained from the **Statics** program. **Determinant** is used to calculate the determinant of a Jacobian matrix. This determinant is called the manipulability measure and it expresses the manipulability of the robot quantitatively [Yoshikawa 1984].

To obtain concrete answers, of course, the system should utilize the GDCC ability to approximate the real roots of univariate equations.

6.2 Constructing the Voronoi Diagram

We developed an application program which constructs *Voronoi Diagram* written in GDCC.

By using the constraint paradigm, we can make a program without describing a complicated algorithm. A Voronoi diagram can be constructed by using constraints which describe only the properties or the definition of the Voronoi diagram. This program can compute the Voronoi polygon of each point in parallel.

6.2.1 Definition of the Voronoi Diagram

For a given finite set of points S in a plane, a Voronoi diagram is a partition of the plane so that each region of the partition is a set of points in the plane closer to a point in S in the region than to any other points in S [Preparata and Shamos 1985].

In the simplest case, the distance between two points is defined as the Euclidian distance. In this case, a Voronoi diagram is defined as follows.

Given a set S of N points in the plane, for each point P_i in S, the *Voronoi polygon* denoted as $V(P_i)$ is defined by the following formula.

$$V(P_i) = \{P \mid d(P, P_i) < d(P, P_j), \forall j \neq i\}$$

where $d(P, P_i)$ is a Euclidian distance between P and P_i.

The Voronoi diagram is a partition so that each region is the Voronoi polygon of each point (see Figure 9). The vertices of the diagram are *Voronoi vertices* and its line segments are *Voronoi edges.*

Voronoi diagrams are widely used in various application areas, such as physics, ecology and urbanology.

6.2.2 Detailed Design

The methods of constructing Voronoi diagrams are classified into the following two categories:

1. The incremental method([Green and Sibson 1978]), and

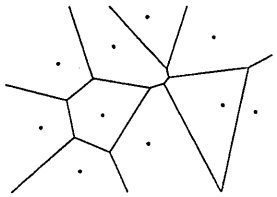


Figure 9: A Voronoi Diagram

2. The divide-and-conquer method([Shamos and Hoey 1975]).

However, the simplest approach to constructing a Voronoi diagram is, of course, constructing its polygons one at a time.

Given two points, P_i and P_j, a set of points closer to P_i than to P_j is just a half-plane containing P_i that is divided by the perpendicular bisector of $\overline{P_iP_j}$. We name this line $H(P_i, P_j)$.

The Voronoi polygon of P_i can be obtained by the following formula.

$$V(P_i) = \cap_{j \neq i} H(P_i, P_j) \cdot \text{p}$$

By using the linear constraint solver for GDCC, the Voronoi polygon can be constructed by the following algorithm which utilizes the above method to obtain the polygon directly.

```
E_s ← {x ≥ 0, y ≥ 0, x ≤ x_Max, y ≤ y_Max}
{loop a}
for i = 1 to n
    CF_0 ← linear_constraint_solver(E_s)
    for j = 1 to n
        if(j ≠ i) then
          E_j ← y ≤ (P_jx − P_ix)/(P_jy − P_iy) · x
                  +(P_jy^2 + P_ix^2 − P_iy^2 − P_jx^2)/2 · (P_jy − P_iy)^2
          CF_j ← linear_constraint_solver(E_j ∪ CF_{j−1})
          Let {eq_1, eq_2, ..., eq_k}(0 ≤ k ≤ n) be
          a set of equations obtained by changing
          inequality symbols in CF_j to equation symbols.
    {loop b}
    for l = 1 to k
        vertices := {}
        m := 1
        while (m =< k &
                number of elements of vertices ≠ 2)
                pp ← intersection(eq_l, eq_m)
                if pp satisfies the constraint set CF_i
                  then vertices := {pp} ∪ vertices
                m := m + 1
        add the line segment between vertices
                to Voronoi edges.
end.
```

In this algorithm, the first half computes the Voronoi polygon for each point's P_i by obtaining all perpendicular bisectors of segments between P_i and other points and eliminating redundant ones. The second half computes the Voronoi edges.

[2]This inequality represents a half plane divided by a perpendicular bisector of $\overline{(P_i,\ P_j)}$

Table 4: Runtime and reductions

Points	Processors					Reductions
	1	2	4	8	15	(×1000)
10	130	67	33	17	16	5804
	1	1.936	3.944	7.377	7.844	
20	890	447	241	123	88	42460
	1	1.990	3.685	7.218	10.077	
50	4391	2187	1102	566	336	210490
	1	2.007	3.981	7.749	13.065	
100	17287	8578	4305	2191	1263	830500
	1	2.015	4.014	7.887	13.679	
200	52360	26095	13028	6506	3500	2458420
	1	2.006	4.018	8.047	14.959	
400	220794	110208	54543	27316	14819	10161530
	1	2.003	4.048	8.082	14.899	

To realize the above algorithm on parallel processors, each procedure for each i in loop a in the above is assigned to a group of processes. That is, there are n process groups. Each procedure for each l in the loop b is assigned to a process in the same process group. This means that each process group contain k processes. These $n \times k$ processes are mapped onto multi-processor machines.

6.2.3 Results

Table 4 shows the execution time and speedup for 10 to 400 points with 1 to 15 processors.

According to the results, we can conclude that, when the number of points is large enough, we can obtain efficiency which is almost in proportion to the number of processors.

By using this algorithm, we can handle the problem of constructing a Voronoi diagram in a very straight forward manner. Actually, comparing the size of the programs, this algorithm can be described in almost one third of the size of the program that is used by the incremental method.

7 Conclusion

In the FGCS project, we developed two CLP languages: CAL, and GDCC to establish the knowledge programming environment, and to write application programs. The aim of our research is to construct a powerful high-level programming language which is suitable for knowledge processing. It is well known that constraints play an important role in both knowledge representation and knowledge processing. That is, CLP is a promising candidate as a high level programming language in this field.

Compared with other CLP languages such as CLP($\mathcal{R}$), Prolog III, and CHIP, we can summarize the features of CAL and GDCC as follows:

- CAL and GDCC can deal with nonlinear algebraic constraints.

- In the algebraic constraint solver, the approximate values of all possible real solutions can be computed, if there are only finite number of solutions.
- CAL and GDCC have a multiple environment handler. Thus, even if there is more than one answer constraints, users can manipulate them flexibly.
- Users can use multiple constraint solvers, and furthermore, users can define and implement their own constraint solvers.

CAL and GDCC enable us to write possibly nonlinear polynomial equations on complex numbers, relations on truth values, relations on sets and their elements, and linear equations and linear inequalities on real numbers.

Since starting to write application programs for the algebraic constraint solver in the field of handling robot, we have wanted to compute the real roots of univariate nonlinear polynomials. We made this possible with CAL by adding a function to approximate the real roots, and we modified the Buchberger algorithm able to handle approximation values.

Then, we faced the problem that a variable may have more than one value. To handle this situation in the framework of logic programming, we introduced a context tree in CAL. In GDCC, we introduced blocks into the language specification. The block in GDCC not only handle multiple values, but also localize the failure of constraint solvers.

As for CAL, the following issues are still to be considered:

1. Meta facilities:
   Users cannot deal with a context tree from a program, that is, meta facilities in CAL are insufficient to allow users to do all possible handling of answer constraints themselves.
2. Partial evaluation of CAL programs:
   Although we try to analyze constraint sets by adopting dependency analysis, that work will be more effective when combined with partial evaluation technology or abstract interpretation.
3. More application programs:
   We still have a few application programs in CAL. By writing many application programs in various application field, we will have ideas to realize a more powerful CLP language. For this purpose, we are now implementing CAL in a dialect of ESP, called Common ESP, which can run on the UNIX operating system to be able to use CAL in various machines.

As for GDCC, the following issues are still to be considered:

1. Handling multiple contexts:
   Although current GDCC has functionalities to handle multiple contexts, users have to express everything explicitly. Therefore, we can design high-level tools to handle multiple contexts in GDCC's language specification.
2. More efficient constraint solvers:
   We need to improve both the absolute performance and the parallel speedup of the constraint solvers.
3. More application programs:
   Since parallel CLP language is quite new language, writing application programs may help us to make it powerful and efficient.

Considering our experiences of using CAL and GDCC and the above issues, we will refine the specification and the implementation of GDCC.

These refinements and experiments on various application programs clarified the need for a sufficiently efficient constraint logic programming system with high functionalities in the language facilities.

Acknowledgment

The research on the constraint logic programming system was carried out by researchers in the fourth research laboratory in ICOT in tight cooperation with cooperating manufactures and members of the CLP working group. Our gratitude is first due to those who have given continuous encouragement and helpful comments. Above all, we particularly thank Dr. K. Fuchi, the director of the ICOT research center, Dr. K. Furukawa, a vice director of the ICOT research center, and Dr. M. Amamiya of Kyushu University, the chairperson of the CLP working group.

We would also like to thank a number of researchers we contacted outside of ICOT, in particular, members of the working group for their fruitful and enlightening discussions and comments.

Special thanks go to all researchers in the fourth research laboratory: Dr. K. Sakai, Mr. T. Kawagishi, Mr. K. Satoh, Mr. S. Sato, Dr. Y. Sato, Mr. N. Iwayama, Mr. D. J. Hawley, who is now working at Compuflex Japan, Mr. H. Sawada, Mr. S. Terasaki, Mr. S. Menju, and the many researchers in the cooperating manufacturers.

References

[Aiba *et al.* 1988] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, November 1988.

[Backelin and Fröberg 1991] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor, *Proceedings of ISSAC'91*. ACM, July 1991.

[Benichou *et al.* 1971] M. Benichou, L. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, (1), 1971.

[Boege *et al.* 1986] W. Boege, R. Gebauer, and H. Kredel. Some Examples for Solving Systems of Algebraic Equations by Calculating Gröbner Bases. *Journal of Symbolic Computation*, 2(1):83–98, 1986.

[Borning *et al.* 1989] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the International Conference on Logic Programming*, 1989.

[Buchberger 1985] B. Buchberger. Gröbner bases: An Algorithmic Method in Polynomial Ideal Theory. In N. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publ. Comp., Dordrecht, 1985.

[CAL Manual] Institute for New Generation Computer Technology. *Contrante Avec Logique version 2.12 User's manual.* in preparation.

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proceedings of FGCS'84*, pages 292–298, 1984.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato, and T. Miyazaki. Overview of Parallel Inference Machine Operationg System (PIMOS). In *International Conference on Fifth Generation Computer Systems*, pages 230–251, 1988.

[Clarke *et al.* 1990] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, October 1990.

[Cohen 1990] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7), July 1990.

[Colmerauer 1987] A. Colmerauer. Opening the Prolog III Universe: A new generation of Prolog promises some powerful capabilities. *BYTE*, pages 177–182, August 1987.

[Dincbas *et al.* 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, November 1988.

[Dulmage and Mendelsohn 1963] A. L. Dulmage and N. S. Mendelsohn. Two algorithms for bipartite graphs. *Journal of SIAM*, 11(1), March 1963.

[Green and Sibson 1978] P. J. Green and R. Sibson. Computing Dirichlet Tessellation in the Plane. *The Computer Journal*, 21, 1978.

[Hofmann 1989] C. M. Hoffmann. *Gröbner Bases Techniques*, chapter 7. Morgan Kaufmann Publishers, Inc., 1989.

[Jaffar and Lassez 1987] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *4th IEEE Symposium on Logic Programming*, 1987.

[Kapur and Mundy 1988] K. Kapur and J. L. Mundy. Special volume on geometric reasoning. *Artificial Intelligence*, 37(1-3), December 1988.

[Kutzler 1988] B. Kutzler. *Algebraic Approaches to Automated Geometry Theorem Proving.* PhD thesis, Research Institute for Symbolic Computation , Johannes Kepler University, 1988.

[Lloyd 1984] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Maher 1987] M. J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[Marriott and Sondergaard 1990] K. Marriott and H. Sondergaard. Analysis of constraint logic programs. In *Proc. of NACLP '90*, 1990.

[Menju *et al.* 1991] S. Menju, K. Sakai, Y. Satoh, and A. Aiba. A Study on Boolean Constraint Solvers. Technical Report TM 1008, Institute for New Generation Computer Technology, February 1991.

[Nagai 1991] Y. Nagai. Improvement of geometric theorem proving using dependency analysis of algebraic constraint (in Japanese). In *Proceedings of the 42nd Annual Conference of Information Processing Society of Japan*, 1991.

[Nagai and Hasegawa 1991] Y. Nagai and R. Hasegawa. Structural analysis of the set of constraints for constraint logic programs. Technical report TR-701, ICOT, Tokyo, Japan, 1991.

[Oki *et al.* 1989] H. Oki, K. Taki, S. Sei, and S. Furuichi. Implementation and evaluation of parallel Tsumego program on the Multi-PSI (in Japanese). In *Proceedings of the Joint Parallel Processing Symposium (JSPP'89)*, 1989.

[Ponder 1990] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 51–74. Academic Press, 1990.

[Preparata and Shamos 1985] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[Sakai and Aiba 1989] K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Application. *Journal of Symbolic Computation*, 8:589–603, 1989.

[Saraswat 1989] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[Sato and Aiba 1991] S. Sato and A. Aiba. An Application of CAL to Robotics. Technical Report TM 1032, Institute for New Generation Computer Technology, February 1991.

[Sato and Sakai 1988] Y. Sato and K. Sakai. Boolean Gröbner Base, February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[Sato *et al.* 1991] Y. Sato, K. Sakai, and S. Menju. Solving constraints over sets by Boolean Gröbner bases (in Japanese). In *Proceedings of The Logic Programming Conference'91*, September 1991.

[Satoh 1990] K. Satoh. Formalizing Soft Constraints by Interpretation Ordering. In *Proceedings of 9th European Conference on Artificial Intelligence*, pages 585–590, 1990.

[Satoh and Aiba 1990a] K. Satoh and A. Aiba. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, ICOT, Tokyo, Japan, 1990.

[Satoh and Aiba 1990b] K. Satoh and A. Aiba. Hierarchical Constraint Logic Language: CHAL. Technical Report TR-592, ICOT, Tokyo, Japan, 1990.

[Senechaud 1990] P. Senechaud. Implementation of a parallel algorithm to compute a Gröbner basis on Boolean polynomials. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[Shamos and Hoey 1975] M. I. Shamos and D. Hoey. Closest-point problems. In *Sixteenth Annual IEEE Symposium on Foundations of Computer Sceience*, 1975.

[Siegl 1990] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[Takano 1986] M. Takano. Design of robot structure (in Japanese). *Journal of Japan Robot Society*, 14(4), 1986.

[Terasaki *et al.* 1992] S. Terasaki, D. J. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama, and A. Aiba. Parallel Constraint Logic Programming Language GDCC and its Parallel Constarint Solvers. In *International Conference on Fifth Generation Computer Systems*, 1992.

[Tohyama 1989] S. Tohyama. *Robotics for Machine Engineer (in Japanese).* Sougou Denshi Publishing Corporation, 1989.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 33(6), December 1990.

[Vidal 1990] J. P. Vidal. The Computation of Gröbner bases on a shared memory multi-processor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

[van Emden and Kowalski 1976] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4), October 1976.

[Van Hentenryck 1989] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip with pepsys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Yoshikawa 1984] T. Yoshikawa. Measure of manipulatability of robot arm (in Japanese). *Journal of Japan Robot Society*, 12(1), 1984.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Theorem Provers and Their Applications

Ryuzo Hasegawa and Masayuki Fujita

Fifth Research Laboratory
Institute for New Generation Computer Technology
4-28 Mita 1-chome, Minato-ku, Tokyo 108, Japan
{hasegawa, mfujita}@icot.or.jp

Abstract

This paper describes the results of the research and development of automated reasoning systems(ARS) being conducted by the Fifth Research Laboratory at ICOT. The major result was the development of a parallel theorem proving system MGTP (Model Generation Theorem Prover) in KL1 on a parallel inference machine, PIM. Currently, we have two versions of MGTP. One is MGTP/G, which is used for dealing with ground models. The other is MGTP/N, used for dealing with non-ground models. With MGTP/N, we have achieved a more than one-hundred-fold speedup for condensed detachment problems on a PIM/m consisting of 128 PEs. Nonmonotonic reasoning and program synthesis are taken as promising and concrete application area for MGTP provers. MGTP/G is actually used to develop legal reasoning systems in ICOT's Seventh Research Laboratory. Advanced inference and learning systems are studied for expanding both reasoning power and application areas. Parallel logic programming techniques and utility programs such as 'meta-programming' are being developed using KL1. The technologies developed are widely used to develop applications on PIM.

1 Introduction

The final goal of the Fifth Generation Computer Systems (FGCS) project was to realize a knowledge information processing system with intelligent user interfaces and knowledge base systems on parallel inference machines. A high performance and highly parallel inference mechanism is one of the most important technologies to come out of our pursuit of this goal.

The major goal of the Fifth Research Laboratory, which is conducted as a subgoal of the problem-solving programming module of FGCS, is to build very efficient and highly parallel automated reasoning systems (ARS) as advanced inference systems on parallel inference machines (PIM), taking advantage of the KL1 language and PIMOS operating system. On ARS we intend to develop application systems such as natural language processing, intelligent knowledgebases, mathematical theorem proving systems, and automated programming systems. Furthermore, we intend to give good feedback to the language and operating systems from KL1 implementations and experiments on parallel inference hardware in the process of developing ARS.

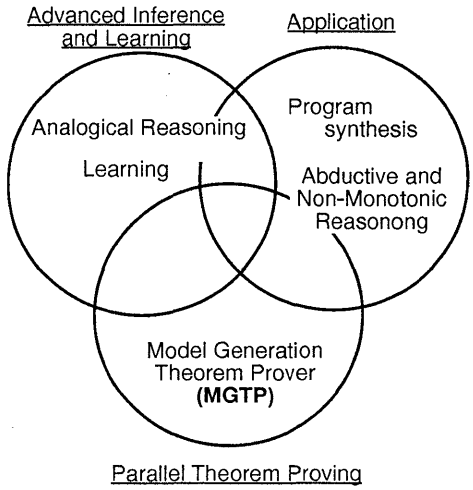


Figure 1: Goals of Automated Reasoning System Research at ICOT

We divided ARS research and development into the following three goals (Figure 1):

(1) Developing Parallel Theorem Proving Technologies on PIM
Developing very efficient parallel theorem provers on PIM by squeezing the most out of the KL1 language is the major part of this task. We have concentrated on the model generation method, whose inference mechanism is based on hyper-resolution. We decided to develop two types of model generation theorem provers to cover ground non-Horn problems and non-ground Horn problems. To achieve maximum performance on PIM, we have focused on the

technological issues below:

(a) Elimination of redundant computation
Eliminating redundant computation in the process of model generation with the least overhead is an important issue. Potential redundancy lies in conjunctive matching at hyper-resolution steps or in the case splitting of ground non-Horn problems.

(b) Saving time and space by eliminating the over generation of models
For the model generation method, which is based on hyper-resolution as a bottom-up process, over generation of models is an essential problem of time and space consumption. We regard the model generation method as generation and test procedures and have introduced a controlling mechanism called *Lazy Model Generation.*

(c) Finding PIM-fitting and scalable parallel architecture
PIM is a low communication cost MIMD machine. Our target is to find a parallel architecture for model generation provers, which draws the maximum power from PIM. We focused on OR parallel architecture to exploit parallelism in the case splitting of a ground non-Horn prover, MGTP/G, and on AND parallel architecture to exploit parallelism in conjunctive matching and subsumption tests of a non-ground Horn prover, MGTP/N.

One of the most important aims of developing theorem provers in KL1 is to draw the maximum advantage of parallel logic programming paradigms from KL1. Programming techniques developed in building theorem provers help to, or are commonly used to, develop various applications, such as natural language processing systems and knowledge base systems, on the PIM machines based on logic programming and its extension. We focused on developing meta-programming technology in KL1 as a concrete base for this aim. We think it is very useful to develop broader problem solving applications on PIM and to extend KL1 to support them.

(2) Application
A model generation theorem prover has a general reasoning power in various AI areas because it can simulate the widely applied tableaux method effectively. Building an efficient analytic tableaux prover for modal propositional logic on model generation theorem provers was the basic goal of this extension. This approach could naturally be applied to abductive reasoning in AI systems and logic programming with negation as failure linked with broader practical AI applications such as diagnosis.

We focused on automated programming as one of the major application areas for theorem provers in the non-Horn logic systems, in spite of difficulty. There has been a long history of program synthesis from specifications in formal logic. We aim to make a first-order theorem prover that will act as a strong support tool in this approach. We have set up three different ways of program construction: realizability interpretation in the constructive mathematics to generate functional programs, temporal propositional logic for protocol generation, and the Knuth-Bendix completion technique for interface design of concurrent processes in Petri Net. We stressed the experimental approach in order to make practical evaluation.

- Advanced Inference and Learning
Theorem proving technologies themselves are rather saturated in their basic mechanisms. In this subgoal, extension of the basic mechanism from deductive approach to analogical, inductive, and transformational approaches is the main research target. Machine learning technologies on logic programs and meta-usage of logic are the major technologies that we decided to apply to this task.

  By using analogical reasoning, we intended to formally simulate the intelligent guesswork that humans naturally do, so that results could be obtained even when deductive systems had no means to deduce to obtain a solution because of incomplete information or very long deductive steps.

  Taking the computational complexity of inductive reasoning into account, we elaborated the learning theories of logic programs by means of predicate invention and least-general generalization, both of which are of central interest in machine learning.

  In transformational approach, we used fold/unfold transformation operations to generate new efficient predicates in logic programming.

The following sections describe these three tasks of research on automated reasoning in ICOT's Fifth Research Laboratory for the three years of the final stage of ICOT.

2 Parallel Theorem Proving Technologies on PIM

In this section, we describe the MGTP provers which run on Multi-PSI and PIM. We present the technical essence of KL1 programming techniques and algorithms that we developed to improve the efficiency of MGTP.

2.1 Parallel Model Generation Theorem Prover MGTP

The research on parallel theorem proving systems aims at realizing highly parallel advanced inference mechanisms that are indispensable in building intelligent knowledge information systems. We started this research project on parallel theorem provers about two and a half years ago. The immediate goal of the project is to develop a parallel automated reasoning system on the parallel inference machine, PIM, based on KL1 and PIMOS technology. We aim at applying this system to various fields such as intelligent database systems, natural language processing, and automated programming.

At the beginning, we set the following as the main subjects.

- To develop very fast first-order parallel theorem provers
  As a first step for developing KL1-technology theorem provers, we adopted the model generation method on which SATCHMO is based as a main proof mechanism. Then we implemented a model-generation based theorem prover called MGTP. Our reason was that the model generation method is particularly suited to KL1 programming as explained later. Based on experiences with the development of MGTP, we constructed a "TP development support system" which provided us with useful facilities such as a proof tracer and a visualizer to see the dynamic behavior of the prover.

- To develop applications
  Although a theorem prover for first-order logic has the potential to cover most areas of AI, it has not been so widely used as Prolog. One reason for this is the inefficiency of the proof procedure and the other is lack of useful applications. However, through research on program synthesis from formal specification[Hasegawa *et al.*, 1990], circuit verification, and legal reasoning[Nitta *et al.*, 1992], we became convinced that first-order theorem provers can be effectively used in various areas. We are now developing an automated program synthesis system, a specification description system for exchange systems, and abductive and non-monotonic reasoning systems on MGTP.

- To develop KL1 programming techniques
  Accumulating KL1 programming techniques through the development of theorem provers is an important issue. We first developed KL1 compiling techniques to translate given clauses to corresponding KL1 clauses, thereby achieving good performance for ground clause problems. We also developed methods to parallelize MGTP by making full use of logical variables and the stream data type of KL1.

- To develop KL1 meta-programming technology
  This is also an important issue in developing theorem provers. This issue is discussed in Section 2.1.2 Meta-Programming in KL1. We have implemented basic meta-programming tools called *Meta-Library* in KL1. The meta-library is a collection of KL1 programs which offers routines such as full unification, matching, and variable managements.

2.1.1 Theorem Prover in KL1 Language

Recent developments in logic programming have made it possible to implement first-order theorem provers efficiently. Typical examples are PTTP by Stickel [Stickel 1988], and SATCHMO by Manthey and Bry [Manthey and Bry 1988].

PTTP is a backward-reasoning prover based on the model elimination method. It can deal with any first-order formula in Horn clause form without loss of completeness and soundness.

SATCHMO is a forward-reasoning prover based on the model generation method. It is essentially a hyper-resolution prover, and imposes a condition called range-restricted on a clause so that we can derive only ground atoms from ground facts. SATCHMO is basically a forward-reasoning prover but also allows backward-reasoning by employing Prolog over the Horn clauses.

The major advantage of these systems is because the input clauses are represented with Prolog clauses and most parts of deductions can be performed through normal Prolog execution.

In addition to this method we considered the following two alternative implementations of object-level variables in KL1:

(1) representing object-level variables with KL1 ground terms

(2) representing object-level variables with KL1 variables

The first approach might be the right path in meta-programming where object- and meta-levels are separated strictly, thereby giving it clear semantics. However, it forces us to write routines for unification, substitution, renaming, and all the other intricate operations on variables and environments. These routines would become considerably larger and more complex than the main program, and introduce overhead to orders of magnitude.

In the second approach, however, most of operations on variables and environments can be performed beside the underlying system instead of running routines on top of it. Hence, it enables a meta-programmer to save writing tedious routines as well as gaining high efficiency. Furthermore, one can also use Prolog var predicates to write routines such as occurrence checks in order to make built-in unification sound, if necessary. Strictly speaking, this approach may not be chosen since it makes the

distinction between object- and meta-level very ambiguous. However, from a viewpoint of program complexity and efficiency, the actual profit gained by the approach is considerably large.

In KL1, however, the second approach is not always possible, as in the Prolog case. This is because the semantics of KL1 never allows us to use a predicate like Prolog var. In addition, KL1 built-in unification is not the same as Prolog's counterpart, in that unification in the guard part of a KL1 clause is limited to one way and a unification failure in the body part is regarded as a semantic error or exception rather than as a failure which merely causes backtrack in Prolog. Nevertheless, we can take the second approach to implement a theorem prover where ground models are dealt with, by utilizing the features of KL1 as much as possible.

Taking the above discussions into consideration, we decided to develop both the MGTP/G and MGTP/N provers so that we can use them effectively according to the problem domain being dealt with.

The ground version, MGTP/G, aims to support finite problem domains, which include most problems in a variety of fields, such as database processing and natural language processing.

For ground model cases, the model generation method makes it possible to use just matching, rather than full unification, if the problem clauses satisfy the *range-restrictedness* condition [1].

This suggests that it is sufficient to use KL1's head unification. Thus we can take the KL1 variable approach for representing object-level variables, thereby achieving good performance.

The key points of KL1 programming techniques developed for MGTP/G are as follows: (Details are described in the next section.)

- First, we translate a given set of clauses into a corresponding set of KL1 clauses. This translation is quite simple.
- Second, we perform conjunctive matching of a literal in a clause against a model element by using KL1 head unification.
- Third, at the head unification, we can automatically obtain fresh variables for a different instance of the literal used.

The non-ground version, MGTP/N, supports infinite problem domains. Typical examples are mathematical theorems, such as group theory and implicational calculus.

For non-ground model cases, where full unification with occurrence check is required, we are forced to follow the KL1 ground terms approach. However, we do not necessarily have to maintain variable-binding pairs as processes in KL1. We can maintain them by using the vector facility supported by KL1, as is often used in ordinary language processing systems. Experimental results show that vector implementation is several hundred times faster than process implementation.

In this case, however, we cannot use the programming techniques developed for MGTP/G. Instead, we have to use a conventional technique, that is, interpreting a given set of clauses instead of compiling it into KL1 clauses.

[1] A clause is said to be range-restricted if every variable in the clause has at least one occurrence in its antecedent.

Problems	Solutions	Tools
1 Redundancy in Conjunctive Matching	Ramified Stack MERC	KL1 programming techniques
2 Unification/ Subsumption	Term Indexing	+
3 Irrelevant Clauses	Partial Falsify Relevancy Test	Firmware Coding
4 Meta-Programming in KL1	Meta-Library	
5 Overgeneration of Models	Lazy Model Generation	+
6 Parallelism		PIM machine
Non-Horn Ground	OR Parallel / And Sequential	
Horn	AND Parallel	

Figure 2: Major Problems and Technical Solutions

2.1.2 Key Technologies to Improve Efficiency

We developed several programming techniques in the process of seeking ways to improve the efficiency of model generation theorem provers. Figure 2 shows a list of the problems which prevented good performance and the solutions we obtained. In the following sections we outline the problems and their solutions.

Redundancy in Conjunctive Matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching. Let us consider a clause having two antecedent literals, and suppose we have a model candidate M at some stage i in the

proof process. To perform conjunctive matching of an antecedent literal in the clause against a model element, we need to pick all possible pairs of atoms from M. Imagine that we are to extend M with a model-extending atom Δ, which is in the consequent of the clause, but not in M. Then in the next stage, we need to pick pairs of atoms from $M \cup \Delta$. The number of pairs amounts to:

$$(M \cup \Delta)^2 = M \times M \cup M \times \Delta \cup \Delta \times M \cup \Delta \times \Delta.$$

However, doing this in a naive manner would introduce redundancy. This is because $M \times M$ pairs were already considered in the previous stage. Thus we must only choose pairs which contain at least one Δ.

(1) RAMS Method

The key point of the RAMS (Ramified Stack) method is to retain in a literal instance stack the intermediate results obtained in conjunctive matching. They are instances which are a result of matching a literal against a model element. This algorithm exactly computes a repeated combination of Δ and an atom picked from M without duplication([Fujita and Hasegawa 1990]).

For non-Horn clause cases, the literal instance stack expands a branch every time case splitting occurs, and grows like a tree. This is how the RAMS name was derived. Each branch of the tree represents a different model candidate.

The ramified-stack method not only avoids redundancy in conjunctive matching but also enables us to share a common model. However, it has one drawback: it tends to require a lot of memory to retain intermediate literal instances.

(2) MERC Method

The MERC (Multi-Entry Repeated Combination) method([Hasegawa 1991]) tries to solve the above problem in the RAMS method. This method does not need a memory to retain intermediate results obtained in the conjunctive matching. Instead, it needs to prepare $2^n - 1$ clauses for the given clause having n literals as its antecedent.

The outline of the MERC method is shown in Figure 3. For a clause having three antecedent literals, $A_1, A_2, A_3 \rightarrow C$, we prepare seven clauses, each of which corresponds to a repeated combination of Δ and M, and perform the conjunctive matching using the combination pattern. For example, a clause corresponding to a combination pattern $[M, \Delta, M]$ first matches literal A_2 against Δ. If the match succeeds, the remaining literals, A_1 and A_3, are matched against an element picked out of M. Note that each combination pattern includes at least one Δ, and that the $[M, M, M]$ pattern is excluded.

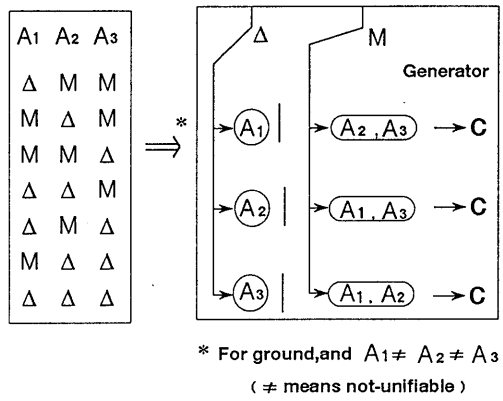


Figure 3: Multiple-Entry Repeated Combination (MERC) Method

There are some trade-off points between the RAMS method and the MERC method. In the RAMS method, every successful result of matching a literal A_i against model elements is memorized so as not to rematch the same literal against the same model element. On the other hand, the MERC method does not need such a memory to store the information of partial matching. However, it still contains a redundant computation. For instance, in the computation for $[M, \Delta, \Delta]$ and $[M, \Delta, M]$ patterns, the common subpattern, $[M, \Delta]$, will be recomputed. The RAMS method can remove this sort of redundancy.

Speeding up Unification/Subsumption

Almost all computation time is used in the unification and subsumption tests in the MGTP. Term indexing is a classical way and the only way to improve this process to one-to-many unification/subsumption. We used the discrimination tree as the indexing structure.

Figure 4 shows the effect of Term Memory on a typical problem on MGTP/G.

Optimal use of Disjunctive Clauses

Loveland et. al. [Wilson and Loveland 1989] indicated that irrelevant use of disjunctive clauses in the ground model generation prover rises useless case splitting, thereby leads to serious redundant searches. Arbitrary selected disjunctive clauses in MGTP may lead to a combinatorial explosion of redundant models. An artificial yet suggestive example is shown in Figure 5.

In MGTP/G, we developed two methods to deal with this problem. One method is to introduce upside-down

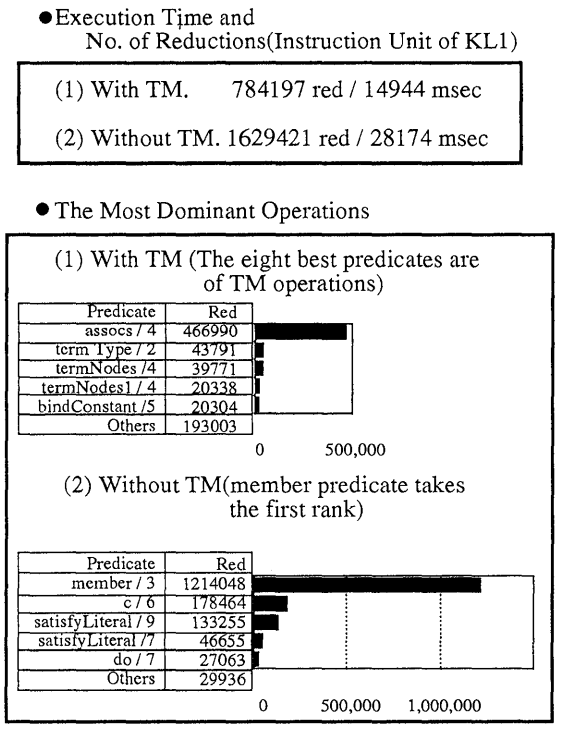


Figure 4: Speed up by Term Memory

$$
\begin{aligned}
\texttt{false} &:- \texttt{p(c,X,Y)} && (1)\\
\texttt{false} &:- \texttt{q(X,c,Y)} && (2)\\
\texttt{false} &:- \texttt{r(X,Y,c)} && (3)\\
\texttt{s(a)} & && (4)\\
\texttt{s(b)} & && (5)\\
\texttt{s(c)} & && (6)\\
\texttt{s(X), s(Y), s(Z)} &\rightarrow \\
\texttt{p(X,Y,Z); q(X,Y,Z); r(X,Y,Z)} & && (7)
\end{aligned}
$$

Figure 5: Example Problem to Relevancy Testing

meta-interpretation(UDMI)[Stickel 1991] into MGTP/G. By using upside-down meta-interpretation, the above problem was compiled into the bottom-up rules in Figure 6.

Note that this is against the range restricted rule but is safe with Prolog-unification.

The other method is to keep the positive disjunctive clauses obtained by the process of reasoning. False checks are made independently on each literal in the disjunctive model elements with unit models and if the check succeeds then that literal is eliminated. The disjunctive models can be sorted by their length. This method showed considerable speed-up for n-queens problems and enumeration of finite algebra.

$$
\begin{aligned}
\texttt{true} &\rightarrow \texttt{gp(c,X,Y).} && (1-1)\\
\texttt{gp(c,X,Y), p(c,X,Y)} &\rightarrow \texttt{false.} && (1-2)\\
\texttt{true} &\rightarrow \texttt{gq(X,c,Y).} && (2-1)\\
\texttt{gq(X,c,Y), q(X,c,Y)} &\rightarrow \texttt{false.} && (2-2)\\
\texttt{true} &\rightarrow \texttt{gr(X,Y,c).} && (3-1)\\
\texttt{gr(X,Y,c), r(X,Y,c)} &\rightarrow \texttt{false.} && (3-2)\\
\texttt{true} &\rightarrow \texttt{s(a)} && (4)\\
\texttt{true} &\rightarrow \texttt{s(b)} && (5)\\
\texttt{true} &\rightarrow \texttt{s(c)} && (6)\\
\texttt{s(X), s(Y), s(Z),} & \\
\texttt{gp(X,Y,Z), gq(X,Y,Z), gr(X,Y,Z)} &\rightarrow \\
\texttt{p(X,Y,Z); q(X,Y,Z); r(X,Y,Z)} & && (7)
\end{aligned}
$$

Figure 6: Compiled code in UDMI

Meta-Programming in KL1

Developing fast meta-programs such as unification and matching programs is very significant in making a prover efficient. Most parts of proving processes are the executions of such programs. The efficiency of a prover depends on how efficient meta-programs are made.

In Prolog-Technology Theorem Provers such as PTTP and SATCHMO, object-level variables[2] are directly represented by Prolog variables. With this representation, most operations on variables and environments can be performed beside the underlying system Prolog. This means that we can gain high efficiency by using the functions supported by Prolog. Also, a programmer can use the Prolog var predicate to write routines such as occurrence checks in order to make built-in unification sound, if such routines are necessary.

Unfortunately in KL1, we cannot use this kind of technique. This is because:

1) the semantics of KL1 never allow us to use a predicate like var,

2) KL1 built-in unification is not the same as its Prolog counterpart in that unification in the guard part of a KL1 clause can only be one-way, and

3) a unification failure in the body part is regarded as a program error or exception that cannot be backtracked.

We should, therefore, treat an object-level variable as constant data (ground term) rather than as a KL1 variable. It forces us to write routines for unification, substitution, renaming, and all the other intricate operations of variables and environments. These routines can become extremely large and complex compared to the main program, and may make the overhead bigger.

To ease the programmer's burden, we developed *Meta-Library*. This is a collection of KL1 programs to support meta-programming in KL1 [Koshimura *et al.*, 1990].

[2]variables appearing in a set of given clauses

The meta-library includes facilities such as full unification with occurrence check, and variable management routines. The performance of each program in the meta-library is relatively good. For example, unification program runs at $0.25 \sim 1.25$ times the speed of built-in unification.

The major functions in meta-library are as follows.

```
unify(X,Y, Env,^NewEnv)
unify_oc(X,Y, Env,^NewEnv)
match(Pattern,Terget, Env,^NewEnv)
oneway_unify(Pattern,Terget, Env,^NewEnv)
copy_term(X,^NewX, Env,^NewEnv)
shallow(X,Env, ^NewNnv)
freeze(X,^FrozenX, Env)
melt(X,^MeltedX, Env)
create_env(^Env, Size)
fresh_var(Env, ^VarAndNewEnv)
equal(X,Y, Env, ^YN)
is_type(X,Env, ^Type)
unbound(X,Env, ^YN)
database(RequestStream)
get_object(KL1Term, ^Object)
get_kl1_term(Object, ^KL1Term)
```

Over-Generation of Models

A more important issue with regard to the efficiency of model generation based provers is reducing the total amount of computation and memory space required in proof processes.

Model-generation based provers must perform the following three operations.

- create new model elements by applying the model extension rule to the given clauses using a set of model-extending atoms Δ and a model candidate set M (model extension).
- make a subsumption test for a created atom to check if it is subsumed by the set of atoms already being created, usually by the current model candidate.
- make a false check to see if the unsubsumed model element derives false by applying the model rejection rule to the tester clauses (rejection test).

The problem with the model generation method is the huge growth in the number of generated atoms and in the computational cost in time and space, which is incurred by the generation processes. This problem becomes more critical when dealing with harder problems which require deeper inferences (longer proofs), such as Lukasiewicz problems.

To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes, and that generation should be performed only when required by the test.

Table 1: Comparison of complexities (for unit tester clause)

Algorithm	T	S	G	M
Basic	ρm^2	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
Full-test/Lazy	ρm^2	$\mu m^{2\alpha}$	m^2	ρm^2
Lazy & Lookahead	m^2	$(\mu/\rho)m^\alpha$	m/ρ	m

† m is the number of elements in a model candidate when *false* is detected in the basic algorithm.

‡ ρ is the survival rate of a generated atom, μ is the rate of successful conjunctive matchings ($\rho \leq \mu$), and $\alpha(1 \leq \alpha \leq 2)$ is the efficiency factor of a subsumption test.

For this we proposed a lazy model generation algorithm[Hasegawa *et al.*, 1992] that can reduce the amount of computation and space necessary for obtaining proofs.

Table 1 compares the complexities of the model generation algorithms[3], where T(S/G) represents the number of rejection tests (subsumption tests/model extensions), and M represents the number of atoms stored.

From a simple analysis, it is estimated that the time complexity of the model extension and subsumption test decreases from $O(m^4)$ in the algorithms without lazy control to $O(m)$ in the algorithms with lazy control. For details, refer to [Hasegawa *et al.*, 1992].

Parallelizing MGTP

There are three major sources when parallelizing the proving processes in the MGTP prover: multiple model candidates in a proof, multiple clauses to which model generation rules are applied, and multiple literals in conjunctive matching.

Let us assume that the prime objective of using the model generation method is to find a model as a solution. There may be alternative solutions or models for a given problem. We take it as OR-parallelism to seek these multiple solutions at the same time.

According to our assumption, multiple model candidates and multiple clauses are taken as sources for exploiting OR-parallelism. On the other hand, multiple literals are the source of AND-parallelism since all the literals in a clause relate to a single solution, where shared variables in the clause should have compatible values.

For ground non-Horn cases, it is sufficient to exploit OR parallelism induced by case splitting. For Horn clause cases, we have to exploit AND parallelism. The

[3]The basic algorithm taken by OTTER[McCune 1990] generates a bunch of new atoms before completing rejection tests for previously generated atoms. The full-test algorithm completes the tests before the next generation cycle, but still generates a bunch of atoms each time. Lookahead is an optimization method for testing wider spaces than in Full-test/Lazy.

main source of AND parallelism is conjunctive matching. Performing subsumption tests in parallel is also very effective for Horn clause cases.

In the current MGTP, we have not yet considered the non-ground and non-Horn cases.

(1) **Parallelization of MGTP/G**

With the current version of the MGTP/G, we have only attempted to exploit OR parallelism on the Multi-PSI machine.

(a) Processor allocation

The processor allocation methods that we adopted achieve 'bounded-OR' parallelism in the sense that OR-parallel forking in the proving process is suppressed so as to meet restricted resource circumstances.

One way of doing this, called *simple allocation*, is sketched as follows. We expand model candidates starting with an empty model using a single master processor until the number of candidates exceeds the number of available processors, then distribute the remaining tasks to slave processors. Each slave processor explores the branches assigned without further distributing tasks to any other processors. This simple allocation scheme for task distribution works fairly well since communication costs can be minimized.

(b) Speed-up on Multi-PSI

One of the examples we used is the N-queens problem given below.

$$
\begin{aligned}
C_1 :&\quad true \rightarrow p(1,1); p(1,2); \ldots; p(1,n).\\
&\quad \ldots\\
C_n :&\quad true \rightarrow p(n,1); p(n,2); \ldots; p(n,n).\\
C_{n+1} :&\quad p(X_1,Y_1), p(X_2,Y_2),\\
&\qquad unsafe(X_1,Y_1,X_2,Y_2)\\
&\qquad\qquad \rightarrow false.
\end{aligned}
$$

The first N clauses express every possible placing of queens on an N by N chess board. The last clause expresses the constraint that a pair of queens must satisfy. So, the problem would be solved when either one model (one solution) or all the models (all solutions) are obtained for the clause set. The performance has been measured on an MGTP/G prover running on a Multi-PSI using the simple allocation method stated above.

The speedup obtained using up to 16 processors are shown in Figure 7. For the 10-queens problem, almost linear speedup is obtained as the number of processors increases. The speedup

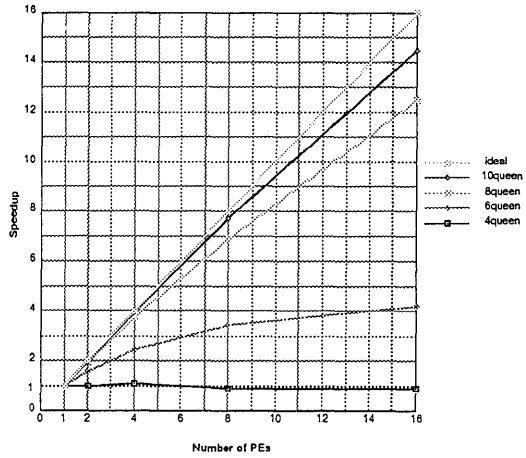


Figure 7: Speedup of MGTP/G on Multi-PSI (N-queens)

rate is rather small for the 4-queens problem only. This is probably because in such a small problem, the constant amount of interpretation overhead would dominate the proper tasks for the proving process.

(2) **Parallelization of MGTP/N**

For MGTP/N, we have attempted to exploit AND parallelism for Horn problems.

We have several choices when parallelizing model-generation based theorem provers:

1) proofs which change or remain unchanged according to the number of PEs used
2) model sharing (copying in a distributed memory architecture) or model distribution, and
3) master-slave or master-less.

A proof changing prover may achieve super-linear speedup while a proof unchanging prover can achieve, at most, linear speedup.

The merit of model sharing is that time consuming subsumption testing and conjunctive matching can be performed at each PE independently with minimal inter-PE communication. On the other hand, the benefit of model distribution is that we can obtain memory scalability. The communication cost, however, increases as the number of PEs increases, since generated atoms need to flow to all PEs for subsumption testing.

The master-slave configuration makes it easy to build a parallel system by simply connecting a sequential version of MGTP/N on a slave PE to the master PE. However, it needs to be designed with devices so as to minimize the load on the master

Table 2: Performance of MGTP/N (Th 5 and Th 7)

Problem		16 PEs	64 PEs
Th5	Time (sec)	41725.98	11056.12
	Reductions	38070940	40759689
	KRPS/PE	57.03	57.60
	Speedup	1.00	3.77
Th7	Time (sec)	48629.93	13514.47
	Reductions	31281211	37407531
	KRPS/PE	40.20	43.25
	Speedup	1.00	3.60

process. On the other hand, a master-less configuration, such as a ring connection, allows us to achieve pipeline effects with better load balancing, whereas it becomes harder to implement suitable control to manage collaborative work among PEs.

Our policy in developing parallel theorem provers is that we should distinguish between the speedup effect caused by parallelization and the search-pruning effect caused by strategies. In the proof changing parallelization, changing the number of PEs is merely betting, and may cause the strategy to be changed badly even though it results in the finding of a shorter proof.

Given the above, we implemented a proof unchanging version of MGTP/N in a master-slave configuration based on lazy model generation. In this system, generator and subsumption processes run in a demand-driven mode, while tester processes run in a data-driven mode. The main features of this system are as follows:

1) Proof unchanging allows us to obtain greater speedup as the number of PEs increases;
2) By utilizing the synchronization mechanism supported by KL1, sequentiality in subsumption testing is minimized;
3) Since slave processes spontaneously obtain tasks from the master and the size of each task is well equalized, good load balancing is achieved;
4) By utilizing the stream data type of KL1, demand driven control is easily and efficiently implemented.

By using the demand driven control, we can not only suppress unnecessary model extensions and subsumption tests but also maintain a high running rate that is the key to achieving linear speedup.

Figure 8 displays the speedup ratio for condensed detachment problems #3, #58, and #77, taken from[McCune and Wos 1991], by running the

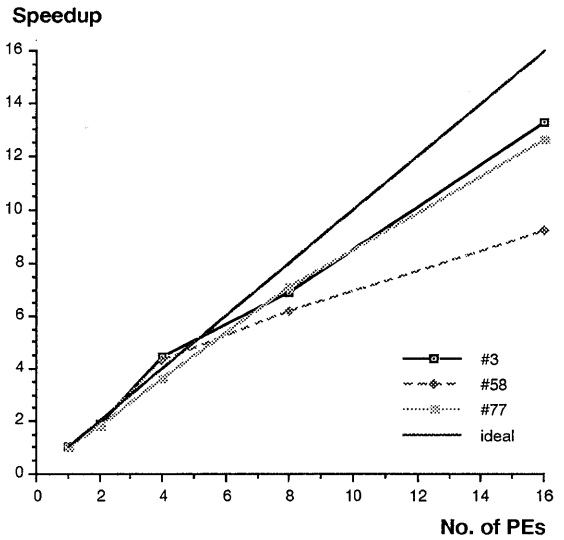


Figure 8: Speedup ratio

MGTP/N prover on Multi-PSI using 16PEs. The execution times taken to solve these problems are 218, 12, and 37 seconds. As shown in the figure, there is no saturation in performance up to 16 PEs and greater speedup is obtained for the problems which consume more time.

Table 2 shows the performance obtained by running MGTP/N for Theorems 5 and 7 [Overbeek 1990], which are also condensed detachment problems, on Multi-PSI with 64 PEs. We did not use heuristics such as sorting, but merely limited term size and eliminated tautologies. Full unification is written in KL1, which is thirty to one hundred times slower than that written in C on SUN/3s and SPARCs. Note that the average running rate per PE for 64 PEs is actually a little higher than that for 16 PEs. With this and other results, we were able to obtain almost linear speedup.

Recently we obtained a proof of Theorem 5 on PIM/m with 127 PEs in 2870.62 sec and nearly 44 billion reductions (thus 120 KRPS/PE). Taking into account the fact that the PIM/m CPU is about twice as fast as that of Multi-PSI, we found that almost linear speedup can be achieved, at least up to 128 PEs.

2.2 Reflection and Parallel Meta-Programming System

Reflection is the capability to feel the current state of the computation system or to dynamically modify it. The form of *reflection* we are interested in is the *computational reflection* proposed by [Smith 1984]. We try to

incorporate meta-level computation and computational reflection in logic programming language in a number of directions.

As a foundation, a reflective sequential logic language *R-Prolog** has been proposed [Sugano 1990]. This language allows us to deal with syntactic and semantic objects of the language itself legally by means of several coding operators. The notion of computational reflection is also incorporated, which allows computational systems to recognize and modify their own computational states. As a result, some of the extra-logical predicates in Prolog can be redefined in a consistent framework. We have also proposed a reflective parallel logic programming language RGHC (Reflective Guarded Horn Clauses)[Tanaka and Matono 1992]. In RGHC, a *reflective tower* can be constructed and collapsed in a dynamic manner, using *reflective predicates*. A prototype implementation of RGHC has also been performed. It seems that RGHC is unique in the simplicity of its implementation of *reflection*. The meta-level computation can be executed at the same speed as its object-level computation. we also try to formalize distributed reflection, which allows concurrent execution of both object level and meta level computations [Sugano 1991]. The scope of reflection is specified by grouping goals that share local environments. This also models the eventual publication of constraints.

We have also built up several application systems based on meta-programming and reflection. These are the experimental programming system ExReps [Tanaka 1991], the process oriented GHC debugger [Maeda *et al.*, 1990, Maeda 1992] and the strategy management shell [Kohda and Maeda 91a, Kohda and Maeda 1991b].

ExReps is an experimental programming environment for parallel logic languages, where one can input programs and execute goals. It consists of an abstract machine layer and an execution system layer. Both layers are constructed using meta-programming techniques. Various *reflective operations* are implemented in these layers.

The process oriented GHC debugger provides high-level facilities, such as displaying processes and streams in tree views. It can control a the behavior of a process by interactively blocking or editing its input stream data. This makes it possible to trace and check program execution from a programmer's point of view.

A strategy management shell takes charge of a database of load-balancing strategies. When a user job is input, the current leading strategy and several experimental alternative strategies for the job are searched for in the database. Then the leading task and several experimental tasks of the job are started. The shell can evaluate the relative merits between the strategies, and decides on the leading strategy for the next stage when the tasks have terminated.

3 Applications of Automated Reasoning

ARS has a wider application area if connected with logic programming and a formal approach to programming. We extended MGTP to cover modal logic. This extension has lead to abductive reasoning in AI systems and logic programming with negation as failure linked with broader practical applications such as fault diagnostics and legal reasoning. We also focused on programming, particularly parallel programs, as one of the major application area of formal logic systems in spite of difficulties. There has been a long history of program synthesis from specifications in formal logic. We are aiming to make ARS, the foundational strength of this approach.

3.1 Propositional Modal Tableaux in MGTP

MGTP's proof method and the tableaux proof procedure[Smullyan 1968] are very close in computationally. Each rule of tableaux is represented by an input clause for MGTP in a direct manner. In other words, we can regard the input clauses for MGTP as a tableaux implementation language, as Horn clauses are a programming language for Prolog.

MGTP tries to generate a model for a set of clauses in a bottom-up manner. When MGTP successfully generates a model, it is found to be satisfiable. Otherwise, it is found to be unsatisfiable.

$$apply(MGTP, ASetOfClauses) = \begin{cases} satisfiable \\ unsatisfiable \end{cases}$$

Since we regard MGTP as an inference system, a propositional modal tableaux[Fitting 1983, Fitting 1988] has been implemented in MGTP.

$$apply(MGTP, TableauxProver(Formula)) = \begin{cases} satisfiable \\ unsatisfiable \end{cases}$$

In tableaux, a close condition is represented by a negative clause, an input formula by a positive clause and a decomposition rule by a mixed clause for MGTP in a direct manner[Koshimura and Hasegawa 1991].

There are two levels in this prover. One is the MGTP implementation level, the other is the tableaux implementation level. The MGTP level is the inference system level at which we mainly examine speedup of inference such as redundancy elimination and parallelization. At the tableaux level, inference rules, which indicate the property of a proof domain, are described. It follows that we mainly examine the property of the proof domain at the tableaux level. It is useful and helpful to have these two levels, as we can separate the description for the property of the domain from the description for the inference system.

3.2 Abductive and Nonmonotonic Reasoning

Modeling sophisticated agents capable of *reasoning with incomplete information* has been a major theme in AI. This kind of reasoning is not only an advanced mechanism for intelligent agents to cope with some particular situations but an intrinsically necessary condition to deal with commonsense reasoning. It has been agreed that neither human beings nor computers can have all the information relevant to mundane or everyday situations. To function without complete information, intelligent agents should draw some unsound conclusions, or augment theorems, by applying such methods as closed-world assumptions and *default reasoning*. This kind of reasoning is *nonmonotonic*: it does not hold that the more information we have, the more consequences we will be aware of. Therefore, this inference has to anticipate the possibility of later revisions of beliefs.

We treat reasoning with incomplete information as a reasoning system with hypotheses, or *hypothetical reasoning* [Inoue 1988], in which a set of conclusions may be expanded by incorporating other hypotheses, unless they are contradictory. In hypothetical reasoning, inference to reach the best explanations, that is, computing hypotheses that can explain an observation, is called *abduction*. The notion of explanation has been a fundamental concept for various AI problems such as diagnoses, synthesis, design, and natural language understanding. We have investigated methodologies of hypothetical reasoning from various angles and have developed a number of abductive and nonmonotonic reasoning systems.

Here, we shall present hypothetical reasoning systems built upon the MGTP [Fujita and Hasegawa 1991]. The basic idea of these systems is to translate formulas with special properties, such as nonmonotonic provability (*negation as failure*) and *consistency* of abductive explanations, into some formulas with a kind of modality so that the MGTP can deal with them using classical logic. The extra requirements for these special properties are thus reduced to generate-and-test problems for model candidates. These, can, then, be handled by the MGTP very efficiently through case-splitting of non-unit consequences and rejection of inconsistent model candidates. In the following, we show how the MGTP can be used for logic programs containing negation as failure, and for abduction.

3.2.1 Logic Programs and Disjunctive Databases with Negation as Failure

In recent theories of logic programming and deductive databases, declarative semantics have been given to the extensions of logic programs, where the negation-as-failure operator is considered to be a *nonmonotonic* modal operator. In particular, logic programs or deductive databases containing both negation as failure (*not*) and classical negation ($\neg$) can be used as a powerful knowledge representation tool, whose applications contain reasoning with incomplete knowledge [Gelfond and Lifschitz 1991], default reasoning, and abduction [Inoue 1991a]. However, for these extended classes of logic programs, the top-down approach cannot be used for computation because there is no local property in evaluating programs. For example, there has been *no* top-down proof procedure which is sound with respect to the *stable model semantics* for general logic programs. We thus need bottom-up computation for correct evaluation of negation-as-failure formulas.

In [Inoue *et al.*, 1992a], a bottom-up computation of *answer sets* for any class of function-free logic programs is provided. These classes include the *extended disjunctive databases* [Gelfond and Lifschitz 1991], the proof procedure of which has not been found. In evaluating $not\, P$ in a bottom-up manner, it is necessary to interpret $not\, P$ with respect to a fixpoint of computation because, even if P is not currently proved, P might be proved in subsequent inferences. We thus came up with a completely different way of thinking for *not*. When we have to evaluate $not\, P$ in a current model candidate we split the model candidate in two: (1) the model candidate where P is assumed not to hold, and (2) the model candidate where it is necessary that P holds. Each negation-as-failure formula $not\, P$ is thus translated into negative and positive literals with a modality expressing belief, i.e., "disbelieve P" (written as $\neg \mathsf{K}P$) and "believe P" (written as $\mathsf{K}P$).

Based on the above discussion, we translate any logic program (with negation as failure) into a *positive disjunctive program* (without negation as failure) of which the MGTP can compute the minimal models. The following is an example of the translation of general logic programs. Let Π be a general logic program consisting of rules of the form:

$$A_l \leftarrow A_{l+1}, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n, \quad (1)$$

where, $n \geq m \geq l \geq 0$, $1 \geq l \geq 0$, and each A_i is an atom. Rules without heads are called *integrity constraints* and are expressed by $l = 0$ for the form (1). Each rule in Π of the form (1) is translated into the following MGTP rule:

$$A_{l+1}, \ldots, A_m, \rightarrow \neg \mathsf{K}A_{m+1}, \ldots, \neg \mathsf{K}A_n, A_l \mid \mathsf{K}A_{m+1} \mid \ldots \mid \mathsf{K}A_n. \quad (2)$$

For any MGTP rule of the form (2), if a model candidate S' satisfies $A_{l+1}, \ldots, A_m$, then S' is split into $n - m + l$ ($n \geq m \geq 0$, $0 \leq l \leq 1$) model candidates. Pruning rules with respect to "believed" or "disbelieved" literals are expressed as the following integrity constraints. These are dealt with by using object-level schemata on the MGTP.

$$\neg \mathsf{K}A, A \rightarrow \qquad \text{for every atom } A \quad (3)$$

$$\neg \mathsf{K}A, \mathsf{K}A \rightarrow \qquad \text{for every atom } A \quad (4)$$

Given a general logic program Π, we denote the set of rules consisting of the two schemata (3) and (4) by $tr(\Pi)$,

and the MGTP rules obtained by replacing each rule (1) of Π by a rule (2). The MGTP then computes the fixpoint of model candidates, denoted by $\mathcal{M}(tr(\Pi))$, which is closed under the operations of the MGTP. Although each model candidate in $\mathcal{M}(tr(\Pi))$ contains "believed" atoms, we should confirm that every such atom is actually derived from the program. This checking can be done very easily by using the following constraint. Let $S' \in \mathcal{M}(tr(\Pi))$.

$$\text{For every ground atom } A, \text{ if } \mathsf{K}A \in S', \text{ then } A \in S'. \tag{5}$$

Computation by using MGTP is sound and complete with respect to the stable model semantics in the sense that: S is an answer set (or *stable model*) of Π if and only if S is one of the atoms obtained by removing every literal with the operator K from a model candidate S' in $\mathcal{M}(tr(\Pi))$ such that S' satisfies condition (5).

Example: Suppose that the general logic program Π consists of the four rules:

$$\begin{aligned} R &\leftarrow not\, R, \\ R &\leftarrow Q, \\ P &\leftarrow not\, Q, \\ Q &\leftarrow not\, P. \end{aligned}$$

These rules are translated to the following MGTP rules:

$$\begin{aligned} &\rightarrow \neg\mathsf{K}R, R \mid \mathsf{K}R, \\ &Q \rightarrow R, \\ &\rightarrow \neg\mathsf{K}Q, P \mid \mathsf{K}Q, \\ &\rightarrow \neg\mathsf{K}P, Q \mid \mathsf{K}P. \end{aligned}$$

In this example, the first MGTP rule can be further reduced to

$$\rightarrow \mathsf{K}R.$$

if we prune the first disjunct by the schema (3). Therefore, the rule has computationally the same effect as the integrity constraint:

$$\leftarrow not\, R.$$

This integrity constraint says that every answer set has to contain R: namely, R *should be derived.* Now, it is easy to see that $\mathcal{M}(tr(\Pi)) = \{ S_1, S_2, S_3 \}$, where $S_1 = \{\mathsf{K}R, \neg\mathsf{K}Q, P, \mathsf{K}P\}$, $S_2 = \{\mathsf{K}R, \mathsf{K}Q, \neg\mathsf{K}P, Q, R\}$, and $S_3 = \{\mathsf{K}R, \mathsf{K}Q, \mathsf{K}P\}$. The only model candidate that satisfies the condition (5) is S_2, showing that $\{Q, R\}$ is the unique stable model of Π. Note that $\{P\}$ is not a stable model because S_1 contains $\mathsf{K}R$ but does not contain R.

In [Inoue *et al.*, 1992a], a similar translation was also given to extended disjunctive databases which contain classical negation, negation as failure and disjunctions. Our translation method not only provides a simple fixpoint characterization of answer sets, but also is very helpful for understanding under what conditions each model is stable or unstable. The MGTP can find all answer sets incrementally, without backtracking, and in parallel. The proposed method is surprisingly simple and does not increase the computational complexity of the problem more than computation of the minimal models of positive disjunctive programs. The procedure has been implemented on top of the MGTP on a parallel inference machine, and has been applied to a legal reasoning system.

3.2.2 Abduction

There are many proposals for a logical account of *abduction*, whose purpose is to generate query explanations. The definition we consider here is similar to that proposed in [Poole *et al.*, 1987]. Let Σ be a set of formulas, Γ a set of literals and G a closed formula. A set E of ground instances of Γ is an *explanation of* G *from* (Σ, Γ) if

1. $\Sigma \cup E \models G$, and
2. $\Sigma \cup E$ is consistent.

The computation of explanations of G from (Σ, Γ) can be seen as an extension of proof-finding by introducing a set of hypotheses from Γ that, if they could be proved by preserving the consistency of the augmented theories, would complete the proofs of G. Alternatively, abduction can be characterized by a consequence-finding problem [Inoue 1991b], in which some literals are allowed to be hypothesized (or *skipped*) instead of proved, so that new theorems consisting of only those skipped literals are derived at the end of deductions instead of just deriving the empty clause. In this sense, abduction can be implemented by an extension of deduction, in particular of a top-down, backward-chaining theorem-proving procedure. For example, Theorist [Poole *et al.*, 1987] and SOL-resolution [Inoue 1991b] are extensions of the Model Elimination procedure [Loveland 1978].

However, there is nothing to prevent us from using a bottom-up, forward-reasoning procedure to implement abduction. In fact, we developed the abductive reasoning system APRICOT/0 [Ohta and Inoue 1990], which consists of a forward-chaining inference engine and the ATMS [de Kleer 1986]. The ATMS is used to keep track of the results of inference in order to avoid both repeated proofs of subgoals and duplicate proofs among different hypotheses deriving the same subgoals.

These two reasoning styles for abduction have both merits and demerits, which are complementary to each other. Top-down reasoning is directed to the given goal but may result in redundant proofs. Bottom-up reasoning eliminates redundancy but may prove subgoals unrelated to the proof of the given goal. These facts suggest that it is promising to simulate top-down reasoning using

a bottom-up reasoner, or to utilize cashed results in top-down reasoning. This upside-down meta-interpretation [Bry 1990] approach has been attempted for abduction in [Stickel 1991], and has been extended by incorporating consistency checks in [Ohta and Inoue 1992].

We have already developed several parallel abductive systems [Inoue *et al.*, 1992b] using the the bottom-up theorem prover MGTP. We outline four of them below.

1. **MGTP+ATMS** (Figure 9).

   This is a parallel implementation of APRICOT/0 [Ohta and Inoue 1990] which utilizes the ATMS for checking consistency. The MGTP is used as a forward-chaining inference engine, and the ATMS keeps a current set of beliefs **M**, in which each ground atom is associated with some hypotheses. For this architecture, we have developed an upside-down meta-interpretation method to incorporate the top-down information [Ohta and Inoue 1992].

   Parallelism is exploited by executing the parallel ATMS. However, because there is only one channel between the MGTP and the ATMS, the MGTP often has to wait for the results of the ATMS. Thus, the effect of parallel implementation is limited.

2. **MGTP+MGTP** (Figure 10).

   This is a parallel version of the method described in [Stickel 1991]. In addition, consistency is checked by calling another MGTP (*MGTP_2*). In this system, each hypothesis H in Γ is represented by $fact(H, \{H\})$, and each Horn clause in Σ of the form:

   $$A_1 \wedge \ldots \wedge A_n \supset C\,,$$

   is translated into an MGTP rule of the form:

   $$fact(A_1, E_1), \ldots, fact(A_n, E_n) \rightarrow$$

   $$fact(C, \mathsf{cc}(\bigcup_{i=1}^{n} E_i))\,,$$

   where E_i is a set of hypotheses from Γ on which A_i depends, and the function cc is defined as:

   $$\mathsf{cc}(E) = \begin{cases} E & \text{if } \Sigma \cup E \text{ is consistent} \\ \mathsf{nil} & \text{if } \Sigma \cup E \text{ is not consistent} \end{cases}$$

   A current set of beliefs **M** is kept in the form of $fact(A, E)$ representing a meta-statement that $\Sigma \cup E \models A$, but is stored in the inference engine (*MGTP_1*) itself. Each time *MGTP_1* derives a new ground atom, the consistency of the combined hypotheses is checked by *MGTP_2*.

   The parallelism comes from calling multiple *MGTP_2*'s at one time. This system achieves more speed-up than the MGTP+ATMS method. However, since *MGTP_1* is not parallelized, the effect of parallelization depends heavily on how much consistency checking is being performed in parallel at one time.

3. **All Model Generation Method.**

   No matter how good the MGTP+MGTP method might be, the system still consists of two different components. The possibilities for parallelization therefore remain limited. In contrast, model generation methods do not separate the inference engine and consistency checking, but realize both functions in a single MGTP. In such a method, the MGTP is used not only as an inference engine but also as a generate-and-test mechanism so that consistency checks are automatically performed. For this purpose, we can utilize the extension and rejection of model candidates supplied by the MGTP. Therefore, multiple model candidates can be kept in distributed memories instead of keeping one global belief set **M**, as done in the above two methods, thus great amounts of parallelism can be obtained.

   The all model generation method is the most direct way to implement reasoning with hypotheses. For each hypothesis H in Γ, we supply a rule of the form:

   $$\rightarrow H \mid \neg \mathsf{K} H\,, \qquad (6)$$

   where $\neg \mathsf{K} H$ means that H is not assumed to be true in the model. Namely, each hypothesis is assumed either to hold or not to hold. Since this system may generate $2^{|\Gamma|}$ model candidates, the method is often too explosive for several practical applications.

4. **Skip Method.**

   To limit the number of generated model candidates as much as possible, we can use a method to delay the case-splitting of hypotheses. This approach is similar to the processing of negation as failure with the MGTP [Inoue *et al.*, 1992a], introduced in the previous subsection. That is, we do not supply any rule of the form (6) for any hypothesis of Γ, but instead, we introduce hypotheses when they are necessary. When a clause in Σ contains negative occurrences of abducible predicates $H_1, \ldots, H_m$ ($H_i \in \Gamma$, $m \geq 0$) and is in the form:

   $$A_1 \wedge \ldots \wedge A_l \wedge \underbrace{H_1 \wedge \ldots \wedge H_m}_{abducibles} \supset C\,,$$

   we translate it into the following MGTP rule:

   $$A_1, \ldots, A_l \rightarrow$$

   $$H_1, \ldots, H_m, C \mid \neg \mathsf{K} H_1 \mid \ldots \mid \neg \mathsf{K} H_m\,. \qquad (7)$$

   In this translation, each hypothesis in the premise part is *skipped* instead of being resolved, and is moved to the right-hand side. This operation is

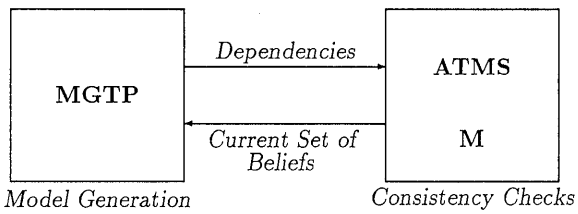


Figure 9: MGTP+ATMS

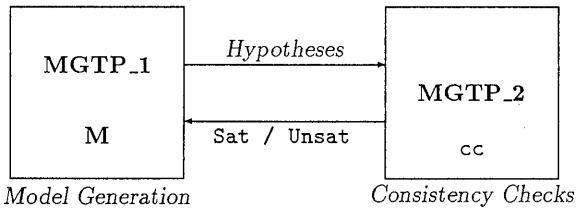


Figure 10: MGTP+MGTP

a counterpart to the **Skip** rule in the top-down approach defined in [Inoue 1991b]. Just as in schema (3) for negation as failure, a model candidate containing both H and $\neg \mathsf{K}H$ is rejected by the schema:

$$\neg \mathsf{K}H,\ H \rightarrow \qquad \text{for every hypothesis } H\,.$$

Some results of evaluation of these abductive systems as applied to planning and design problems are described in [Inoue *et al.*, 1992b]. We are now improving their performance for better parallelism. Although we need to investigate further how to avoid possible combinatorial explosion in model candidate construction for the skip method, we conjecture that the skip method (or some variant thereof) will be the most promising from the viewpoint of parallelism. Also, the skip method may be easily combined with negation as failure so that knowledge bases can contain both abducible predicates and negation-as-failure formulas as in the approach of [Inoue 1991a].

3.3 Program Synthesis by Realizability Interpretation

3.3.1 Program Synthesis by MGTP

We used *Realizability Interpretation* (an extension of *Curry-Howard Isomorphism*) in the area of constructive mathematics [Howard 1980], [Martin 1982] in order to give an executable meaning to proofs obtained by efficient theorem provers.

Our approach for combining prover technologies and *Realizability Interpretation* has the following advantages:

- This approach is prover independent and all provers are possibly usable.
- *Realizability Interpretation* has a strong theoretical background.
- *Realizability Interpretation* is general enough to cover concurrent programs.

Two systems MGTP and PAPYRUS, developed in ICOT, are used for the experiments on sorting algorithms in order to get practical insights into our approach(Figure 11).

A model generation theorem prover (MGTP) implemented in KL1 runs on a parallel machine:Multi-PSI. It searches for proofs of specification expressed as logical formulae. MGTP is a hyper-resolution based bottom up (infers from premises to goal) prover. Thanks to KL1 programming technology, MGTP is simple but works very efficiently if problems satisfy the *range-restrictedness* condition. The inference mechanism of MGTP is similar to SATCHMO[Manthey and Bry 1988], in principle. Hyper-resolution has an advantage for program synthesis in that the inference system is constructive. This means that no further restriction is needed to avoid useless searching.

PAPYRUS (PArallel Program sYnthesis by Reasoning Upon formal Systems) is a cooperative workbench for formal logic. This system handles the proof trees of user defined logic in *Edinburgh Logical Framework*(*LF*)[Harper *et al.*, 1987]. A typed lambda term in LF represents a proof and a program can be extracted from this term by lambda computation. This system treats programs (functions) as the models of a logical formula by user defined *Realizability Interpretation.* PAPYRUS is an integrated workbench for logic and provides similar functions to PX[Hayashi and Nakano 1988], Nuprl[Constable *et al.*, 1986], and Elf[Pfenning 1988].

We faced two major problems during research process:

- Program extraction from a proof in clausal form, and

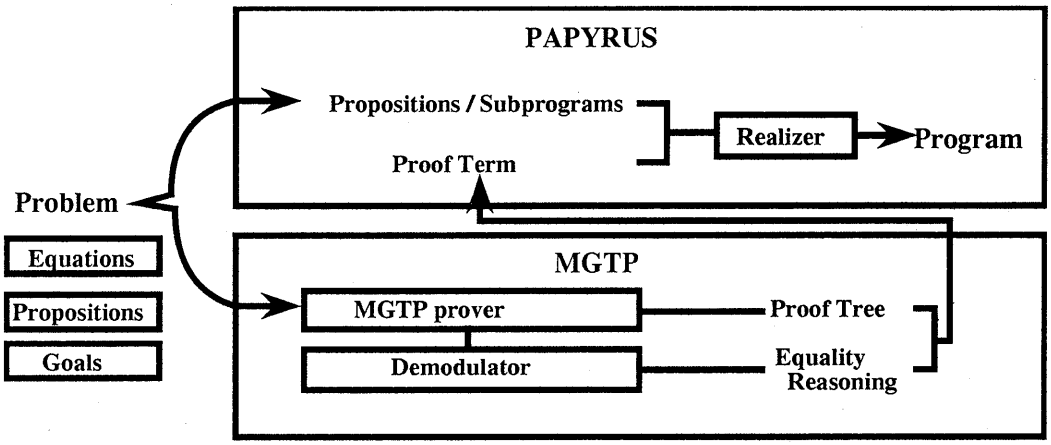


Figure 11: Program Synthesis by MGTP

- Incorporation of induction and equality.

The first problem relates to the fact that programs cannot be extracted from proofs obtained by using the excluded middle, as done in classical logic. The rules for transforming formulae into clausal form contains such a prohibited process. This problem can be solved if the program specification is given in clausal form because a proof can be obtained from the clause set without using the excluded middle. The second problem is that all induction schemes are expressed as second-order propositions. In order to handle this, second-order unification will be needed, which still is impractical. However, it is possible to transform a second-order proposition to a first-order proposition if the program domain is fixed.

Proof steps of equality have nothing to do with computation, provers can use efficient algorithms for equality as an attached procedure.

3.3.2 A Logic System for Extracting Interactive Processes

There has been some research [Howard 1980, Martin 1982, Sato 1986] and [Hayashi and Nakano 1988] into program synthesis from constructive proofs. In this method, an interpretation of formulas is defined, and the consistent proof of the formula can be translated into a program that satisfies the interpretation. Therefore we can identify the formula as the specification of the program, proof as programming, and proof checking as program checking. Though this method has many useful points, the definition of a program in this method is only "λ Term (function)". Thus it is difficult to synthesize a program as a parallel process by which computers can communicate with the outside world.

We proposed a new logic μ, that is, a constructive logic extended by introducing new operators μ and $\natural$. The operator μ is a fixpoint operator on formulae. We can express the non-bounded repetition of inputs and outputs with operators μ and $\natural$. Further, we show a method to synthesize a program as a parallel process like CCS[Milner 1989] from proofs of logic μ. We also show the proof of consistency of Logic μ and the validity of the method to synthesize a program.

3.4 Application of Theorem Proving to Specification of a Switching System

We apply a theorem proving technique to the software design of a switching system, whose specifications are modeled by finite state diagrams.

The main points of this project are the following:

1) Specification description language Ack, based on a transition system.

2) Graphical representation in Ack.

3) Ack interpreter by MGTP.

We introduce the protocol specification description language, Ack. It is not necessary to describe all state transitions concretely using Ack, because several state transitions are deduced from one expression by means of theorem proving. Therefore, we can get a complete specification from an ambiguous one.

Ack is based on a transition system (S, s_0, A, T), where S is a set of state, s_0 ($\in$ S) is an initial state, A is a set of actions, and $T(T \subseteq S \times A \times S)$ is a set of transition relations.

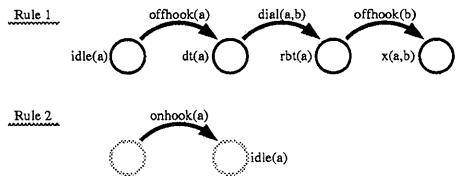


Figure 12: An example of Ack specification

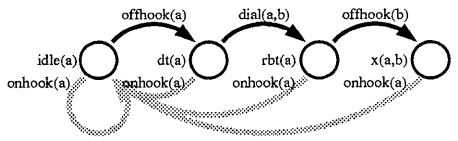


Figure 13: An interpretation result of Ack specification

Graphical representation in Ack consists of labeled circles and arrows. A circle means a state and an arrow means an action. Both have two colors: black and gray. This means that when a gray colored state transition exists a black colored state transition exists.

Textual phrase representation in Ack can be represented by a first order predicate logic by the following.

$$\forall \mathbf{X} \exists \mathbf{Y} (\mathbf{A}[\mathbf{X}] \rightarrow \mathbf{B}[\mathbf{X}, \mathbf{Y}]).$$

where $\mathbf{A}[\mathbf{X}]$ and $\mathbf{B}[\mathbf{X}, \mathbf{Y}]$ are conjunctions of the following atomic formulas.

$\mathbf{state(S)}$ - S means a state.

$\mathbf{trans(A, S_0, S_1)}$ - An action A means a state S_0 to a state S_1.

$\mathbf{A}[\mathbf{X}]$ corresponds to grayed color state transitions and $\mathbf{B}[\mathbf{X}, \mathbf{Y}]$ corresponds to black color state transitions.

The Ack interpreter is described by MGTP. This type of formula is translated into an MGTP formula. A set of models deduced from Ack specification formulae form a complete state transition diagram.

Figure 12 shows an example of Ack specification.

Rule 1 of Figure 1 means the existence of an action sequence from an initial state $idle(a)$ such that $offhook(a) \rightarrow dial(a,b) \rightarrow offhook(b)$. This is represented by the following formula.

$$\begin{aligned} \rightarrow\ & \mathbf{trans(offhook(a), idle(a), dt(a)),} \\ & \mathbf{trans(dial(a, b), dt(a), rbt(a)),} \\ & \mathbf{trans(offhook(b), rbt(a), x(a, b)).} \end{aligned}$$

Rule 2 of Figure 1 means that the action $offhook(a)$ changes *any state* to $idle(a)$. It is represented by the following formula.

$$\begin{aligned} & \forall \mathbf{S}(\mathbf{state(S)} \wedge \mathbf{state(idle(a))} \\ & \rightarrow \mathbf{trans(onhook(a), S, idle(a))}) \end{aligned}$$

Figure 13 shows an interpretation of the result of Figure 12.

In this example, the following four transitions are automatically generated.

- action $onhook(a)$ from $idle(a)$ to $idle(a)$.
- action $onhook(a)$ from $dt(a)$ to $idle(a)$.
- action $onhook(a)$ from $rbt(a)$ to $idle(a)$.
- action $onhook(a)$ from $x(a,b)$ to $idle(a)$.

3.5 MENDELS ZONE: A Parallel Program Development System

MENDELS ZONE is a software development system for parallel programs. The target parallel programming language is MENDEL, which is a textual form of Petri Nets, MENDEL is then translated into the concurrent logic programming language KL1 and executed on the Multi-PSI. MENDEL is regarded as a more user-friendly version of the language. MENDEL is convenient for the programmer to use to design cooperating discrete event systems.

MENDELS ZONE provides the following functions:

1) Data-flow diagram visualizer
[Honiden *et al.*, 1991]

2) Term rewriting system :
Metis[Ohsuga *et al.*, 90][Ohsuga *et al.*, 91]

3) Petri Nets and temporal logic based programming environment
[Uchihira *et al.*, 90a][Uchihira *et al.*, 90b]

For 1), we define the decomposition rule for data-flow diagram and extract the MENDEL component from decomposed data-flow diagrams. A detailed specification process from abstract specification is also defined by a combination of data-flow diagrams and equational formulas.

For 2), Metis is a system to supply experimental environment for studying practical techniques for equational reasoning. The policy of developing Metis is enabling us to implement, test, and evaluate the latest techniques for inference as rapidly and freely as possible. The kernel function of Metis is Knuth-Bendix (KB) completion procedure. We adopt Metis as a tool for verifying the MENDEL component. The MENDEL component can be translated into a component of Petri Nets.

For 3), following sub-functions are provided:

1. Graphic editor

   The designer constructs each component of Petri Nets using the graphic editor, which provides creation, deletion, and replacement. This editor also supports expansion and reduction of Petri Nets.

2. Method editor

   The method editor provides several functions specific to Petri Nets. Using the method editor, the designer describes methods (their conditions and actions) in detail using KL1.

3. Component library

   Reusable component are stored in the component library. The library tool supports browsing and searching for reusable components.

4. Verification and synthesis tool

   Only the skeletons of Petri Nets structures are automatically retracted (slots and KL1 codes of methods are ignored) since our verification and synthesis are applicable to bounded net. The verification tools verifies whether Petri Nets satisfy given temporal logic constraints.

5. Program execution on Multi-PSI

   The verified Petri Nets are translated into their textual form (MENDEL programs). The MENDEL programs are compiled into KL1 programs, which can be executed on Multi-PSI. During execution, firing methods are displayed on the graphic editor, and values of tokens are displayed on the message window. The designer can check visually that program behaves to satisfy his expectation.

4 Advanced Inference and Learning

It is expected that we will, before long, face a *software crisis* in which the necessary quantity of computer software cannot be provided even if we were all to engage in software production. In order to avoid this crisis, it is necessary for a computer system itself to produce software or new information adaptively in problem-solving. The aim of the study on advanced inference and learning is to explore the underlying mechanism for such a system.

In the current stage in which we have no absolute approach to the goal, we have had to do *exhaustive* searches. We have taken three different but co-operative approaches: logical, computational and empirical. In the logical approach, *analogical reasoning* has been analyzed formally and mechanisms for analogical reasoning have been explored. In the computational approach, we have studied *inventing new predicates*, which are one of the most serious problems in learning logic programs. We have also investigated the application of *minimally multiple generalization* for constructive logic pograms learning. In the empirical approach, we have studied automated programming, especially, the *logic program transformation and synthesis* method based on unfold/fold transformation which is a well-known technique for deriving correct and efficient programs.

The following subsections briefly describe these studies and their results.

4.1 Analogical Reasoning

Analogical reasoning is often said to be at the very core of human problem-solving and has long been studied in the field of artificial intelligence. We treat a general type of analogy, described as follows: when two objects, B (called the *base*) and T (called the *target*), share a property S (called the *similarity*), it is conjectured that T satisfies another property P (called the *projected property*) which B satisfies as well.

In the study of analogy, the following have been central problems:

1) Selection of an object as a base w.r.t a target.

2) Selection of pertinent properties for drawing analogies.

3) Selection of a property for projection w.r.t. a certain similarity.

Unfortunately, most previous works were only partially successful in answering these questions, by proposing solutions a priori.

Our objective is to clarify, as formally aspossible,the general relationship between those analogical factors T, B, S, and P under a given theory $\mathcal{A}$. To find the relationship bvetween the analogical factors would answer these problems once and for all. In [Arima 1992, Arima 1991], we clarify such a relation and show a general solution.

When analyzing analogical reasoning formally based on classical logic, the following are shown to be reasonable:

- Analogical reasoning is possible only if a certain form of rule, called the *analogy prime rule* (APR), is a deductive theorem of a given theory. If we let $S(x) = \Sigma(x, S)$ and $P(x) = \Pi(x, P)$, then the rule has the following form:

  $$\forall x, s, p.\ J_{att}(s, p) \wedge J_{obj}(x, s) \wedge \Sigma(x, s) \supset \Pi(x, p),$$

  where each of $J_{att}(s, p)$, $J_{obj}(x, s)$, $\Sigma(x, s)$ and $\Pi(x, p)$ are formulae in which no variable other than its argument occurs freely.

- An analogical conclusion is derived from the APR, together with two particular conjectures: one conjecture is $J_{att}(S, P)$ where, from the information about

the base case, $\Sigma(B,S)$ $(= S(B))$ and $\Pi(B,P)$ $(= P(B))$. The other is $J_{obj}(T,S)$ where, from the information about the target case, $\Sigma(T,S)(= S(T))$.

Also, a candidate based on abduction + deduction is shown for a non-deductive inference system which can yield both conjectures.

4.2 Machine Learning of Logic Programs

Machine Learning is one of the most important themes in the area of artificial intelligence. A learning ability is necessary not only for processing and maintaining a large amount of knowledge information but also for realizing a user-friendly interface. We have studied the invention of new predicates is one of the most serious problems in learning logic programs. We have also investigated the application of minimally multiple generalization to the constructive learning of logic programs.

4.2.1 Predicate Invention

Shapiro's model inference gives a very important strategy for learning programs - an incremental hypothesis search using contradiction backtracing. However, his theory assumes that an initial hypothesis language with enough predicates to describe a target model is given to the learner. Furthermore, it is assumed that the teacher knows the intended model of all the predicates. Since this assumption is rather severe and restrictive, for the practical applications of learning logic programs, it should be removed. To construct a learning system without such assumptions, we have to consider the problem of predicates invention.

Recently, several approaches to this challenging and difficult problem have been presented [Muggleton and Buntine 1988], and [Ling 1989]. However, most of them do not give sufficient analysis on the computational complexity of the learning process, which is where the hypothesis language is growing. We discussed the problem as nonterminal invention in grammatical inference. As is well known, any context-free grammar can be expressed as a special form of the DCG (definite clause grammar) logic program. Thus, nonterminal invention in grammatical inference corresponds to predicate invention.

We have proposed a polynomial time learning algorithm for the class of simple deterministic languages based on nonterminal invention and contradiction backtracking[Ishizaka 1990]. Since the class of simple deterministic languages strictly includes regular languages, the result is a natural extension of our previous work[Ishizaka 1989].

4.2.2 Minimally Multiple Generalization

Another important problem in learning logic programs is to develop a constructive algorithm for learning. Most learning by induction algorithms, such as Shapiro's model inference system, are based on a search or enumerative method. While search and enumerative methods are often very powerful, they are very expensive. A constructive method is usually more efficient than a search method.

In the constructive learning of logic programs, the notion of least generalization [Plotkin 1970] plays a central role. Recently, Arimura proposed a notion of minimally multiple generalization (mmg) [Arimura 1991], a natural extension of least generalization. For example, the pair of heads in a clause in a normal append program is one head in the mmg for the Herbrand model of the program. Thus, mmg can be applied to infer the heads of the target program. Arimura has also given a polynomial time algorithm to compute mmg.

We are now investigating an efficient constructive learning method using mmg.

4.3 Logic Program Transformation / Synthesis

Automated programming is one important advanced inference problem. In researching automatic program transformation and synthesis, the unfold/fold transformation [Burstall and Darlington 1977, Tamaki and Sato 1984] is a well-known program technique to derive correct and efficient programs.

Though unfold/fold rules provide a very powerful methodology for program development, the application of those rules needs to be guided by strategies to obtain efficient programs. In unfold/fold transformation, the efficiency improvement is mainly the result of finding the recursive definition of a predicate, by performing folding steps. Introduction of auxiliary predicates often allows folding steps. Thus, invention of new predicates is one of the most important problems in program transformation.

On the other hand, unfold/fold transformation is often utilized for logic program synthesis. In those studies, unfold/fold rules are used to eliminate quantifiers by folding to obtain definite clause programs from first order formulae. However, in most of those studies, unfold/fold rules were applied nondeterministically and general methods to derive definite clauses were not known.

We have studied logic program transformation and synthesis method based on unfold/fold transformation and have obtained the following results.

(1) We investigated a strategy of logic program transformation based on unfold/fold rules [Kawamura 1991]. New predicates synthesized automatically to perform folding. We also extended

this method to incorporate goal replacement transformation [Tamaki and Sato 1984].

(2) We showed a characterization of classes of first order formulae from which definite clause programs can be derived automatically [Kawamura 1992]. Those formulae are described by Horn clauses extended by universally quantified implicational formulae. A synthesis procedure based on generalized unfold/fold rules [Kanamori and Horiuchi 1987] is given, and with some syntactic restrictions, those formulae successfully transformed into equivalent definite clause programs.

5 Conclusion

We have overviewed research and development of parallel automated reasoning systems at ICOT. The constituent research tasks of three main areas provided us with the following very promising technological results.

(1) Parallel Theorem Prover and its implementation techniques on PIM

We have presented two versions of a model-generation theorem prover MGTP implemented in KL1: MGTP/G for ground models and MGTP/N for non-ground models. We evaluated their performance on the distributed memory multi-processors Multi-PSI and PIM.

Range-restricted problems require only matching rather than full unification, and by making full use of the language features of KL1, excellent efficiency was achieved from MGTP/G.

To solve non-range-restricted problems by the model generation method, however, MGTP/N is restricted to Horn clause problems, using a set of KL1 meta-programming tools called the Meta-Library, to support the full unification and the other functions for variable management.

To improve the efficiency of the MGTP provers, we developed RAMS and MERC methods that enable us to avoid redundant computations in conjunctive matching. We were able to obtain good performance results by using these methods on PSI.

To ease severe time and space requirements in proving hard mathematical theorems (such as condensed detachment problems) by MGTP/N, we proposed the lazy model generation method, which can decrease the time and space complexity of the basic algorithm by several orders of magnitude. Our results show that significant saving in computation and memory can be realized by using the lazy algorithm.

For non-Horn ground problems, case splitting was used as the basic seed of OR parallel MGTP/G. This kind of problem is well-suited to MIMD machine such as Multi-PSI, on which it is necessary to make granularity as large as possible to minimize communication costs. We obtained an almost linear speedup for the n-queens, pigeon hole, and other problems on Multi-PSI, using a simple allocation scheme for task distribution.

For Horn non-ground problems, on the other hand, we had to exploit the AND parallelism inherent to conjunctive matching and subsumption. We found that good performance and scalability were obtained by using the AND parallelization scheme of MGTP/N.

In particular, our latest results, obtained with the MGTP/N prover on PIM/m, showed linear speed-up on condensed detachment problems, at least up to 128 PEs. The key technique is the lazy model generation method, that avoids the unnecessary computation and use of time and space while maintaining a high running rate.

The full unification algorithm, written in KL1 and used in MGTP/N, is one hundred times slower than that written in C on SPARCs. We are considering the incorporation of built-in firmware functions to bridge this gap. But developing KL1 compilation techniques for non-ground models, we believe, will further contribute to parallel logic programming on PIM.

Through the development of MGTP provers, we confirmed that KL1 is a powerful tool for the rapid prototyping of concurrent systems, and that parallel automated reasoning systems can be easily and effectively built on the parallel inference machine, PIM.

(2) Applications

The modal logic prover on MGTP/G realizes two advantages. The first is that the redundancy elimination and parallelization of MGTP/G directly endow the prover with good performance. The second is that direct representation of tableaux rules of modal logic as hyper-resolution clauses are far more suited to adding heuristics for performance. This prover exhibited excellent benchmark results.

The basic idea of non-monotonic and abductive systems on MGTP is to use the MGTP as an meta-interpreter for each system's special properties, such as nonmonotonic provability (*negation as failure*) and the *consistency* of abductive explanations, into formulae having a kind of modality such that MGTP can deal with them within classical logic. The extra requirements for these special properties are thus reduced to "generate-and-test" problems of model candidates that can be efficiently handled by MGTP

through the case-splitting of non-unit consequences and rejection of inconsistent model candidates.

We used MGTP for the application of program synthesis in two ways.

In one approach, we used *Realizability Interpretation*(an extension of *Curry-Howard Isomorphism*), an area of constructive mathematics, to give executable meaning to the proofs obtained by efficient theorem provers.

Two systems, MGTP and PAPYRUS, both developed in ICOT, were used for experiments on sorting algorithms to obtain practical insights into our approach. We performed experiments on sorting algorithms and Chinese Reminder problems and succeeded in obtaining ML programs from MGTP proofs.

To obtain parallel programs, we proposed a new logic μ, that is a constructive logic extended by introducing new operators μ and $\natural$. Operator μ is a fixpoint operator on formulae. We can express the nonbounded repetition of inputs and outputs with operators μ and $\natural$. Furthermore, we showed a method of synthesizing "program" as a parallel process, like CCS, from proofs of logic μ. We also showed the proof of consistency of Logic μ and the validity of the method to synthesize "program".

Our other approach to synthesize parallel programs by MGTP is the use of temporal logic, in which specifications are modeled by finite state diagrams, as follows.

1) Specification description language Ack, based on a transition system.
2) Graphical representation in Ack.
3) Ack interpreter by MGTP.

It is not necessary to describe all state transitions concretely using Ack, because several state transitions are deduced from one expression by theorem proving in temporal logic. Therefore, we can obtain a complete specification from an ambiguous one.

Another approach is to use term rewriting systems(Metis). MENDELS ZONE is a software development system for parallel programs. The target parallel programming language is MENDEL, which is a textual form of Petri Nets, that is translated into the concurrent logic programming language KL1 and executed on Multi-PSI.

We defined the decomposition rules for data-flow diagrams and subsequently extracted programs. Metis provides an experimental environment for studying practical techniques by equational reasoning, of implement, and test. The kernel function of Metis is the Knuth-Bendix (KB) completion procedure. We adopt Metis to verify the components of Petri Nets.

Only the skeletons of Petri Net structures are automatically retracted (slots and the KL1 codes of methods are ignored) since our verification and synthesis are applicable to a bounded net. The verification tool verifies whether Petri Nets satisfy given temporal logic constraints.

(3) Advanced Inference and Learning

To extend the reasoning power of AR systems, we have taken logical, computational, and empirical approaches.

In the logical approach, *analogical reasoning*, considered to be at the very core of human problem-solving, has been analyzed formally and a mechanism for analogical reasoning has been explored. In this approach, our objective was to clarify a general relationship between those analogical factors T, B, S and P under a given theory $\mathcal{A}$, as formally as possible. Determining the relationship between the analogical factors would answer these problems once and for all. We clarified the relationship and formulated a general solution for them all.

In the computational approach, we studied the *inventing of new predicates*, one of the most serious problems in the learning of logic programs. We proposed a polynomial time learning algorithm for the class of simple deterministic languages, based on nonterminal invention and contradiction backtracing. Since the class of simple deterministic languages includes regular languages, the result is a natural extension of our previous work. We have also investigated the application of *minimally multiple generalization* to the constructive learning of logic programs. Recently, Arimura proposed the notion of minimally multiple generalization (*mmg*) . We are now investigating an efficient constructive learning method that uses *mmg*.

In the empirical approach, we have studied automated programming, especially, the *logic program transformation and synthesis* method based on an unfold/fold transformation, a well-known means of deriving correct and efficient programs. We investigated a strategy for logic program transformation based on unfold/fold rules. New predicates are synthesized automatically to perform folding. We also extended this method to incorporate a goal replacement transformation.

We also showed a characterization of the classes of first order formulae, from which definite clause programs can be derived automatically. These formulae are described by Horn clauses, extended by universally quantified implicational formulae. A synthesis procedure based on generalized unfold/fold rules

is given, and with some syntactic restrictions, these formulae can be successfully transformed into equivalent definite clause programs.

These results contribute to the development of FGCS, not only in AI applications, but also in the foundation of the parallel logic programming that we regard as being the kernel of FGCS.

Acknowledgment

The research on automated reasoning systems was carried out by the Fifth Research Laboratory at ICOT in tight cooperation with five manufactures. Thanks are firstly due to who have given support and helpful comments, including Dr. Kazuhiro Fuchi, the director of ICOT, and Dr. Koichi Furukawa, the deputy director of ICOT. Many fruitful discussions took place at the meetings of Working Groups: PTP, PAR, ANR, and ALT. We would like to thank the chair persons and all other members of the Working Groups. Special thanks go to many people at the cooperating manufacturers in charge of the joint research programs.

References

[Fuchi 1990] K. Fuchi, Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Hasegawa *et al.*, 1990] R. Hasegawa, H. Fujita and M. Fujita, A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, in *Italy-Japan-Sweden Workshop '90,* ICOT TR-588, 1990.

[Fujita and Hasegawa 1990] H. Fujita and R. Hasegawa, A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm, ICOT TR-606, 1990.

[Hasegawa 1991] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan, in Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[Hasegawa *et al.*, 1992] R. Hasegawa, M. Koshimura and H. Fujita, Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers, ICOT TR-751, 1992.

[Koshimura *et al.*, 1990] M. Koshimura, H. Fujita and R. Hasegawa, Meta-Programming in KL1, ICOT-TR-623, 1990 (in Japanese).

[Manthey and Bry 1988] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, illinois,* 1988.

[Nitta *et al.*, 1992] K. Nitta, Y. Ohtake, S. Maeda, M. Ono, H. Ohsaki and K. Sakane, HELIC-II: a legal resoning system on the parallel inference machine, in *Proc. of FGCS'92,* Tokyo, 1992.

[Stickel 1988] M.E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, in *Journal of Automated Reasoning* 4 pp.353-380, 1988.

[McCune 1990] W.W. McCune, OTTER 2.0 Users Guide, Argonne National Laboratory, 1990.

[McCune and Wos 1991] W.W. McCune and L. Wos, Experiments in Automated Deduction with Condensed Detachment, Argonne National Laboratory, 1991.

[Overbeek 1990] R. Overbeek, Challenge Problems, (private communication) 1990.

[Wilson and Loveland 1989] D. Wilson and D. Loveland, Incorporating Relevancy Testing in SATCHMO, Technical Report of CS(CS-1989-24), Duke University, 1989.

[Fitting 1983] M. Fitting, *Proof Methods for Modal and Intuitionistic Logic,* D.Reidel Publishing Co., Dordrecht 1983.

[Fitting 1988] M. Fitting, "First-Order Modal Tableaux", *Journal of Automated Reasoning,* Vol.4, No.2, 1988.

[Koshimura and Hasegawa 1991] M. Koshimura and R. Hasegawa, "Modal Propositional Tableaux in a Model Generation Theorem Prover", In *Proceedings of the Logic Programming Conference'91,* Tokyo, 1991 (in Japanese).

[Smullyan 1968] R.M. Smullyan, *First-Order Logic,* Vol 43 of Ergebnisse der Mathematik, Springer-Verlag, Berlin, 1968.

[Arima 1991] J. Arima, A Logical Analysis of Relevance in Analogy, in *Proc. of Workshop on Algorithmic Learning Theory ALT'91,* Japanese Society for Artificial Intelligence, 1991.

[Arima 1992] J. Arima, Logical Structure of Analogy, in *FGCS'92,* Tokyo, 1992.

[Kohda and Maeda 91a] Y. Kohda and M. Maeda, Strategy Management Shell on a Parallel Machine, IIAS RESEARCH Memorandum IIAS-RM-91-8E, Fujitsu, October 1991.

[Kohda and Maeda 1991b] Y. Kohda and M. Maeda, Strategy Management Shell on a Parallel Machine, in poster session of ILPS'91, San Diego, October 1991.

[Maeda *et al.*, 1990] M. Maeda, H. Uoi, N. Tokura, Process and Stream Oriented Debugger for GHC programs, Proceedings of Logic Programming Conference 1990, pp.169-178, ICOT, July 1990.

[Maeda 1992] M. Maeda, Implementing a Process Oriented Debugger with Reflection and Program Transformation, in *Proc. of FGCS'92*, Tokyo, 1992.

[Smith 1984] B.C. Smith, Reflection and Semantics in Lisp, Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

[Sugano 1990] H. Sugano, Meta and Reflective Computation in Logic Programs and its Semantics, Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium, pp.19-34, April, 1990.

[Sugano 1991] H. Sugano, Modeling Group Reflection in a Simple Concurrent Constraint Language, *OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[Tanaka 1991] J. Tanaka, An Experimental Reflective Programming System Written in GHC, *Journal of Information Processing*, Vol.14, No.1, pp.74-84, 1991.

[Tanaka and Matono 1992] J. Tanaka and F. Matono, Constructing and Collapsing a Reflective Tower in Reflective Guarded Horn Clauses, in *Proc. of FGCS'92*, Tokyo, 1992.

[Arimura 1991] H. Arimura, T. Shinohara and S. Otsuki, Polynomial time inference of unions of tree pattern languages. In S. Arikawa, A. Maruoka, and T. Sato, editors, *Proc. ALT '91*, pp. 105–114. Ohmsha, 1991.

[Ishizaka 1989] H. Ishizaka, Inductive inference of regular languages based on model inference. *International journal of Computer Mathematics*, 27:67–83, 1989.

[Ishizaka 1990] H. Ishizaka, Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5(2):151–164, 1990.

[Ling 1989] X. Ling, Inventing theoretical terms in inductive learning of functions - search and constructive methods. In Zbigniew W. Ras, editor, *Methodologies for Intelligent Systems, 4*, pp. 332–341. North-Holland, October 1989.

[Muggleton and Buntine 1988] S. Muggleton and W. Buntine, Machine invention of first-order predicates by inverting resolution. In *Proc. 5th International Conference on Machine Learning*, pp. 339–352, 1988.

[Plotkin 1970] G.D. Plotkin, A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pp. 153–163. Edinburgh University Press, 1970.

[Burstall and Darlington 1977] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", J.ACM, Vol.24, No.1, pp.44–67, 1977.

[Kanamori and Horiuchi 1987] T. Kanamori and K. Horiuchi, "Construction of Logic Programs Based on Generalized Unfold/Fold Rules", Proc. of 4th International Conference on Logic Programming, pp.744–768, Melbourne, 1987.

[Kawamura 1991] T. Kawamura, "Derivation of Efficient Logic Programs by Synthesizing New Predicates", Proc. of 1991 International Logic Programming Symposium, pp.611 – 625, San Diego, 1991.

[Kawamura 1992] T. Kawamura, "Logic Program Synthesis from First Order Logic Specifications", to appear in International Conference on Fifth Generation Computer Systems 1992, Tokyo, 1992.

[Tamaki and Sato 1984] H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs", Proc. of 2nd International Logic Programming Conference, pp.127–138, Uppsala, 1984.

[Bry 1990] F. Bry, Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.

[de Kleer 1986] J. de Kleer, An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[Fujita and Hasegawa 1991] H. Fujita and R. Hasegawa, A model generation theorem prover in KL1 using a ramified-stack algorithm. In: *Proceedings of the Eighth International Conference on Logic Programming* (Paris, France), pp. 535–548, MIT Press, Cambridge, MA, 1991.

[Gelfond and Lifschitz 1991] M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Inoue 1988] K. Inoue, Problem solving with hypothetical reasoning, in *Proc. of FGCS'88*, pp. 1275–1281, Tokyo, 1988.

[Inoue 1991a] K. Inoue, Extended logic programs with default assumptions, in *Proc. of the Eighth International Conference on Logic Programming* (Paris, France), pp. 490–504, MIT Press, Cambridge, MA, 1991.

[Inoue 1991b] K. Inoue, Linear resolution for consequence-finding, To appear in: *Artificial Intelligence*, An earlier version appeared as: Consequence-finding based on ordered linear resolution, in *Proc. of IJCAI-91*, pp. 158–164, Sydney, Australia, 1991.

[Inoue *et al.*, 1992a] K. Inoue, M. Koshimura and R. Hasegawa, Embedding negation as failure into a model generation theorem prover, To appear in *CADE 92*, Saratoga Springs, NY, June 1992.

[Inoue *et al.*, 1992b] K. Inoue, Y. Ohta, R. Hasegawa and M. Nakashima, Hypothetical reasoning systems on the MGTP, ICOT-TR 1992 (in Japanese).

[Loveland 1978] D.W. Loveland, *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[Ohta and Inoue 1990] Y. Ohta and K. Inoue, A forward-chaining multiple context reasoner and its application to logic design, in: *Proceedings of the Second IEEE International Conference on Tools for Artificial Intelligence*, pp. 386–392, Herndon, VA, 1990.

[Ohta and Inoue 1992] Y. Ohta and K. Inoue, A forward-chaining hypothetical reasoner based on upside-down meta-interpretation, in *Proc. of FGCS'92*, Tokyo, 1992.

[Poole *et al.*, 1987] D. Poole, R. Goebel and R. Aleliunas, Theorist: a logical reasoning system for defaults and diagnosis, In: Nick Cercone and Gordon McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pp. 331–352, Springer-Verlag, New York, 1987.

[Stickel 1991]
M.E. Stickel, Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction, ICOT TR-664, 1991.

[Constable *et al.*, 1986] R.L. Constable et al, *Implementing Mathematics with the Nuprl Proof Development System,* Prenticd-Hall, NJ, 1986.

[Hayashi and Nakano 1988] S. Hayashi and H. Nakano, *PX: A Computational Logic,* MIT Press, Cambridge, 1988.

[Harper *et al.*, 1987] R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, in *Symposium on Logic in Computer Science,* IEEE, pp. 194-204, 1987.

[Pfenning 1988] F. Pfenning, Elf: A Language for Logic Definition and Verified Meta-Programming, in *Fourth Annual Symposium on Logic in Computer Science*, IEEE, pp. 313-322, 1989.

[Takayama 1987] Y. Takayama, Writing Programs as QJ Proof and Compiling into Prolog Programs, in *Proc. of IEEE The Symposium on Logic Programming '87*, pp. 278-287, 1987.

[Howard 1980] W.A. Howard, "The formulae-as-types notion of construction", in Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, pp.479-490, 1980.

[Martin 1982] P. Martin-Löf, "Constructive mathematics and computer programming", in Logic, Methodology, and Philosophy of Science VI, Cohen, L.J. et al, eds., North-Holland, pp.153-179, 1982.

[Sato 1986] M. Sato, "QJ: A Constructive Logical System with Types", France-Japan Artificial Intelligence and Computer Science Symposium 86, Tokyo, 1986.

[Milner 1989] R. Milner, "Communication and Concurrency", Prentice-Hall International, 1989.

[Honiden *et al.*, 1990] S. Honiden et al., An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, in *The Journal of Real-Time Systems*, 1990.

[Honiden *et al.*, 1991] S. Honiden et al., An Integration Environment to Put Formal Specification into Practical Use in Real-Time Systems, in *Proc. 6th IWSSD*, 1991.

[Ohsuga *et al.*, 91] A. Ohsuga et al., A Term Rewriting System Generator, in *Software Science and Engineering*, World Scientific ,1991.

[Ohsuga *et al.*, 90] A. Ohsuga et al., Complete E-unification based on an extension of the Knuth-Bendix Completion Procedure, in *Proc. of Workshop on Word Equations and Related Topics*, LNCS 572, 1990.

[Uchihira *et al.*, 90a] N. Uchihira et al., Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, in *Proc. of 23th HICSS*, 1990.

[Uchihira *et al.*, 90b] N. Uchihira et al., Verification and Synthesis of Concurrent Programs Using Petri Nets and Temporal Logic, in *Trans. IEICE*, Vol. E73, No. 12, 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Natural Language Processing Software

Yuichi Tanaka

Sixth Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
ytanaka@icot.or.jp

Abstruct

In the Fifth Generation Computer Systems project, the goal of natural language processing (NLP) is to build an intelligent user interface for the proto-type machine of the Fifth Generation.

In the initial and intermediate stage of our project, mathematical and linguistic theories of discourse understanding was investigated and we built some experimental systems for the theories. In the final stage, we have built a system of general tools for NLP and, using them, developed experimental systems for discourse processing, based on the result and experience of the software development in the past two stages.

In the final stage, we have four themes of NLP research and development.

The first theme, *Language Knowledge-base*, is a collection of basic knowledge for NLP including Japanese grammar and Japanese dictionary. In the second theme, *Language Tool Box*, we have developed several basic tools especially for Japanese processing. Tools are: morphological and syntax analyzers, sentence generator, concordance system, and *etc.* These two themes form the infrastructure of our NLP systems.

Experiment with discourse processing is the third and main theme of our research. We have developed several systems in this field including text generation, discourse structure construction, and dialog systems.

The last theme is parallel processing. We have developed an experimental system for cooperative parallel natural language processing in which morphological analysis, syntax analysis, and semantic analysis are integrated in a uniform process in a type inference framework.

1 Introduction

To establish an intelligent interface between machine and human, it is necessary to research discourse processing. In discourse processing we include not only discourse understanding where compter understands the contents of utterances of human and infers the human's intention,

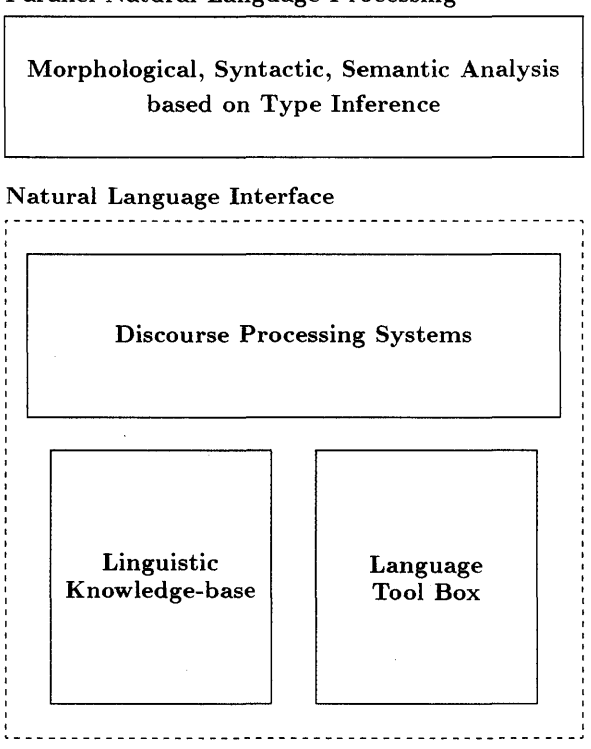


Figure 1: Overview of NLP Software

but also text generation by which more than one sentences expressing speaker's consistent assertion are produced. We put this discourse processing research at the center of our research and development activity, and also develop some supporting tools and data as the infrastructure.

Language Knowledge-base is a collection of basic knowledge for natural language processing including Japanese grammar and Japanese dictionary. We have build a Japanese grammar in phrase structure grammar based on unification grammar formalism. Until now.

there were no Japanese grammar with sufficient size for practical use and usable by every researcher and developer. The purposes of development of this grammar are these two points. It is written in DCG (Definite Clause Grammar) based on the exhaustive investigation of Japanese language phenomena.

Also we have developed a Japanese grammar based on dependency grammar formalism. To reduce ambiguity arisen during analysis, we introduced structural and linguistic constraints on dependency structure based on a new concept 'rank' for each word and word pair.

Adding to the Japanese grammar, we have developed a large-scale Japanese dictionary for morphological analysis. It has about 150,000 entries including more than 40,000 proper nouns so that it can be used for morphological analysis of newspaper articles. These grammar and dictionary are described in section 2.

Language Tool Box is a collection of basic NLP tools especially for Japanese processing. Input and output modules for some experimental NLP systems we made so far, mainly Japanese morphological analyzer, syntax analyzer and sentence generator, were useful for other NLP applications. We have refined their user-interface, made programs robust to unexpected inputs, and increased efficiency to make them easier to apply to various applications.

Currently, not only input and output tools are included in this collection, but also supporting tools for lexicographers and grammar writers such as concordance system and grammar editor. The description of these tools and their publication will be appeared in section 3.

Development of discourse processing systems is the main theme of our research. We have collected rules for language phnomena concerning discourse, and developed several experimental systems in this field including text generation, discourse structure construction, and dialog systems. The text generation system produces one or more paragraphs of text concerning to a given theme based on its belief and judgement. The discourse structure construction system uses discourse rules as a grammar to construct a tree-like discourse structure of a given text. The experimental dialog systems handle user's intention, situation, and position to remove user's misunderstanding and to produce user friendly responces. These system are described in section 4.

As parallel NLP experiment, we have developed a small system for cooperative processing in which morphological analysis, syntax analysis, and semantic analysis are amalgamated into a uniform process in a type inference framework. This system, running on multi-PSI machine, achieves about 12 speed up rate using 32 PEs. Precise description of the system and the experiment will be appeared in section 5.

The overview of the whole activity for these four themes is shown in Figure 1.

2 Linguistic Knowledge-base

Language Knowledge-base is a collection of basic knowledge for natural language processing including Japanese grammar and Japanese dictionary. We have build a Japanese grammar in phrase structure grammar based on unification grammar formalism. There has been no set of standard Japanese grammar rules which people get and handle easily and quickly. This is an obstacle for researchers in Japanese language processing who try to make experimental systems to prove some ideas or who try to build application systems in various field. Our Japanese grammar has been developed to overcome such obstcles and designed as a standard in a sense that it covers most of the general language phenomena and it is written in a common form to various environment, DCG (Definite Clause Grammar). Also we have developed a Japanese grammar based on dependency grammar formalism. Historically, there have been several Japanese dependency grammar because it is recognized easier to build a dependency grammar rules for Japanese because of loose constraints on word order of Japanese language. We introduced structural and linguistic constraints on dependency structure in order to avoid structural ambiguity. These constraints are based on a new concept 'rank' for each word and word pair.

Adding to the Japanese grammar, we have developed a large-scale Japanese dictionary for morphological analysis. It has about 150,000 entries including more than 40,000 proper nouns so that it can be used for morphological analysis of newspaper articles.

The precise description of Language Knowledge-base will be presented in [Sano and Fukumoto 92] submitted to ICOT session of this conference.

2.1 Japanese Grammar

2.1.1 Localized Unification Grammar

Conventional Japanese grammar for computers are not satisfactory to practical application because they lacked formality, uniformity of precision and exhaustiveness [Kuno and Shibatani 89] [Masuoka 89] [Nitta and Masuoka 89].

Having made an exhaustive investigation, we collected language phenomena and rules to explain those phenomena objectively expressed in a DCG style formal description [Pereira 80]. This description is based on the Unification Grammar formalism [Calder 89] [Carlson 89] [Moens 89]. They covers most of the phenomena appearing in contemporary written text [Sano 89] [Sano *et al.* 90] [Sano and Fukumoto 90]. We classified these phenomena according to the complexity of corresponding surface expressions [Sano 91]. Grammar rules are classified also according to their corresponding phnomena. The classification of phenomena (rules) is shown in Table 1.

Table 1: Classification of Grammar Rules

level	phenomena
1~2	single predicate
3~4	negation / aspect / honorification
5	subject+complement+predicate / topicalization
6	passive / causative
7~8	modification (to nouns / to verbs)
9	particles (1) / coordination (2)
10~11	compound sentence / condition
12	particles (2) / coordination (2) / conjunction

The syntactic-semantic structure of sentence is shown in Figure 2. In this figure, State-of-affairs (*SOA*) is the minimum sub-structure of the whole structure. A *SOA* has a predicate with some cases and optional complements. Composition of one or more *SOA*s form a description. The semantic contents of a sentence is a description preceded by a *Topic*. And furthermore the semantics of a sentence contains speaker's intention expressed by *Modal*.

According to this structure, rules of each level (Table1) are divided into several groups. Rules of outermost group analyze speaker's intention through the expression at the end of sentences. Rules of the second group analyze topic-comment structure, that is a dependency relation between a topicalized noun phrase marked by a particle "*wa*" and the main predicate. And rules for analyzing description, voice, *etc.* follow.

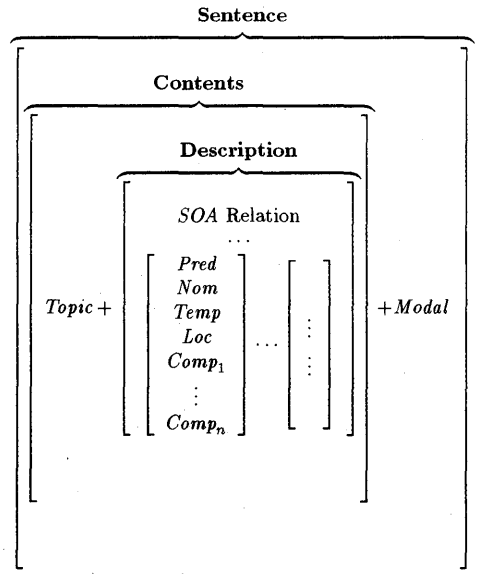


Figure 2: Syntactic-Semantic Structure of a Sentence

An Example of the rules for topic-comment structure will be shown in Figure 3.

$$Cat_0(SYN_2, X_2, \left[topic(X_2, \left\{ \begin{array}{c} SYN_1, \\ REL_1, \\ F_1 \end{array} \right\}) \middle| REL_2 \right], F_2, (X, Z)) \Rightarrow$$
$$Cat_1(SYN_1, X_1, REL_1, F_1, (X, Y)).$$
$$Cat_2(SYN_2, X_2, REL_2, F_2, (Y, Z)).$$

Figure 3: An Example of LUG Grammar Rules

2.1.2 Restricted Dependency Gramar

For Japanese language, there has been many researches on dependency grammar because there are no strong constraints of word order in Japanese [Kodama 87]. In these researches, in order to determine whether a word depends on other, no global information are used but that of only these two words. However, this kind of local information is not sufficient to recognize the structure of whole sentence including topic and ellipsis. Consequently, wrong interpretation of a sentence are produced as a result of dependency analysis [Sugimura and Fukumoto 89].

We introduced structural and linguistic constraints on dependency structure in order to avoid this kind of structural ambiguity. These constraints are described in terms of rank for each word and word pair. Rank represents strength of dependency between words which reflects global information in a whole sentence [Fukumoto and Sano 90]. Definition of ranks and their constraints are described in [Sano and Fukumoto 92] in detail.

Figure 4 shows a structural ambiguity and its resolution. For the sentence "*Kare-ga yobu-to dete-kita.* (When he called ϕ_1, ϕ_2 appeared.)", only the interpretation (a) is adopted because an arc of rank **a** cannot stretch over that of rank **d**.

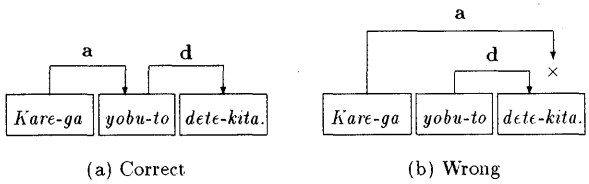


Figure 4: Ambiguity Resolution in RDG Analysis

2.2 Japanese Dictionary

We have developed a concordance system as a tool in *Language Tool Box* (LTB). To serve a huge amount of text data for the concordance system, automatic morphological analysis is necessary. Our large-scale morphological Japanese dictionary has been designed to that purpose.

This Japanese dictionary has about 150.000 entries including more than 40.000 proper nouns so that it can be used for morphological analysis of newspaper articles.

2.3 Software Publication

Japanese grammar and Japanese dictionary stated above will be distributed from ICOT. Japanese grammar in DCG form can be easily installed in any Prolog environment. Japanese dictionary will be distributed with its access method and indexing program which produces TRIE index file for the dictionary entries. Those dictionary programs are written in C.

3 Language Tool Box

Language Tool Box is a collection of basic, general-purpose NLP tools especially for Japanese processing. In the initial and intermediate stage of this project, we developed several experimental systems for discourse understanding so far. As the result of the experiments, the input and output modules for those systems, mainly Japanese morphological analyzer, syntax analyzer and sentence generator, were proved to be useful for other NLP systems. Since then, we have refined their user-interface, made programs robust to unexpected inputs, and increased efficiency to make them easier to apply to various applications.

Currently, not only input and output tools are included in this collection, but also supporting tools for grammar writers and lexicographers such as concordance system with complex key input, browsing / editing / experiment tools for Japanese grammar, and so on.

These software were not applicable for general machines though they were designed general-purpose, because they had been written in ESP, the user language for Personal Sequential Inference Machine PSI. To solve this problem, we transplanted some of these software to CESP (Common ESP) language which was designed as a similar programming language to ESP running on many UNIX workstations.

3.1 Morphological Analysis Tools

Morphological analyzer LAX, located in the front end of LTB, analyzes an unsegmented string of Japanese sentence into a sequence of words and composes semantics of each word from those of morphemes [Kubo *et al.* 88] [Kubo 89] [Sugimura *et al.* 88] [Okumura and Matsumoto 87a] [Okumura and Matsumoto 87b]. It makes use of connectivity matrix which originated from kana-kanji conversion [Aizawa and Ehara 73]. The morpheme dictionary has a TRIE index [Nakajima and Sugimura 89] to improve search speed.

Since there will be, generally, more than one solution for a input sentence in morphological analysis, the most plausible solution is selected by the words minimizing method [Yoshimura *et al.* 82]. The morphology grammar used in this system follows [Morioka 87] and [Sano *et al.* 88].

This system can be also used for developing and extending morphology grammar and dictionary. User interface for that purpose has been deeply considered [Shiraishi *et al.* 90] [Yoneda *et al.* 89].

Configuration of the LAX system is shown in Figure 5 in detail. Total system of this figure is implemented on PSI machine in ESP (Extended Self-contained Prolog). We are now transplanting the system part by part in CESP (Common ESP) to UNIX workstations.

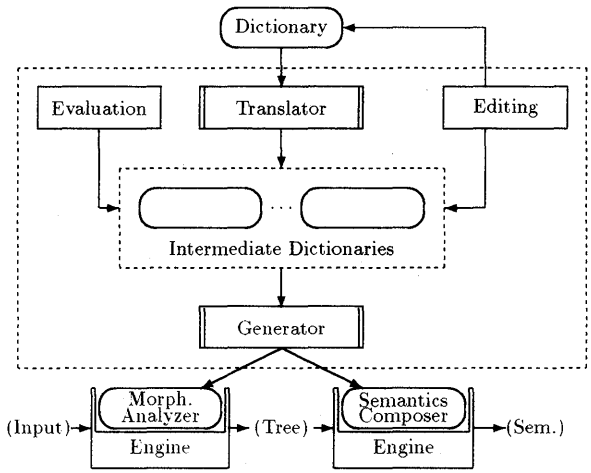


Figure 5: System Configuration of LAX

3.2 Syntax Analysis Tools

Basic algorithm of the syntax analyzer SAX, called AX (Analyzer for syntaX), was first developed in a parallel logic programming language Parlog as a parallel analyzer, then transplanted in GHC [Ueda and Chikayama 90] into parallel analyzer PAX, and in Prolog and ESP into sequencial analyzer SAX [Matsumoto and Sugimura 87] [Okumura and Matsumoto 87a].

The PAX system has been rewritten in KL1 and serves a practical syntax analyzer on Multi-PSI machines [Okumura and Matsumoto 87b] [Satoh 90]. On the other hand, SAX system runs on PSI machine (ESP version) and UNIX workstations (Sicstus-Prolog version; developed at Kyoto University).

3.3 Grammar Writer's Workbench

We have developed a tool for grammar writers. The tool, named LINGUIST, has a simple all-in-one structure described in Fingure 6.

The purpose of this system is to help a grammar writer in evaluation, tracing, and correction of his grammar very easily.

The system has three tools: Generator, Accessor and Debugger. The Generator is a BUP translator [Mat-

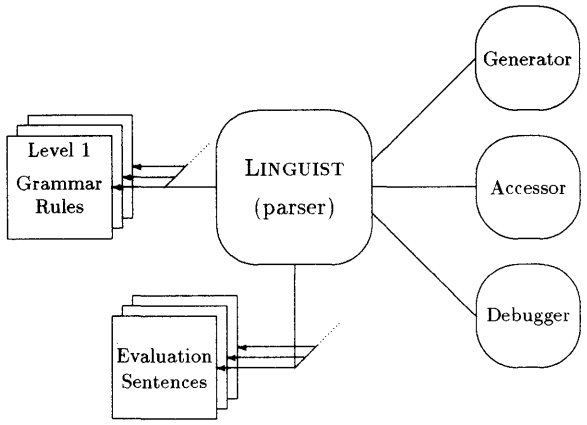


Figure 6: Configuration of LINGUIST System

sumoto *et al.* 83a] [Matsumoto *et al.* 83b] itself which reads a set of grammar rules written in DCG (Definite Clause Grammar) [Pereira 80] and generates a syntax parser. The resulting parser is a core of the system.

The Accessor is a tool for managing linguistic data such as sentences for evaluation, result of analysis (internal representation). One can inspect analysis result with complicatedly nested structure (see section 2.1.1) as a frame or as a graph using structural inspector of the Accessor.

The Debugger contains screen tracer and source level debugger, the former of which displays (partial) syntax tree dynamically with a grammar rule used at that point, and the latter provides correcting function to the source grammar rules at run time.

The LINGUIST system is also transplanted in CESP and, in this case, total system runs on UNIX machines.

3.4 Concordance Tool

When one begin to build a grammar or a dictionary, it is indispensable to collect actual linguistic data from living materials like literature, newspaper and documents.

Concordance or KWIC (Keyword in Context) system is designed for this purpose. It stores large amount of text data and provides searching function on it. When a word or a combination of words is put to the system, it searches text database to retrieve sentences that contain input word(s).

In our concordance system, not only word but also variety of keyword specification are available as input. One can specify compound keyword as

$$k_1\ f_1\ k_2 \cdots f_{n-1}\ k_n$$

where k_i denotes i-th keyword and f_i filler. Fillers, being either definite length (0 or more) or wild card, specify number of words to be discarded between keywords. Keyword can be one of the following or combination of them:

- Surface form (kanji, inflected)
- Root form (kanji, uninflected)
- Reading (kana)
- Part of speech
  - Inflection type
  - Inflected form

One can thus specify a keyword like

{ POS/ verb,
Inflected_form/ *rentai-kei* }.

This system was implemented in ESP on PSI machine at first, then transplanted to CESP.

3.5 Other Tools

There are some more tools in LTB.

CIL is a variation of Prolog. It has frame-like data types (PST; Partially Specified Term) and freeze control structure. In the program segment

```
print(X?),
...
{name/ tanaka, age/ 25} = {age/ X},
...
```

when two PST's are unified, variable X is instantiated, then the freezed term print(X?) is melted to print 25.

The sentence division tool is a one to divide long sentences into the combination of shorter ones to reduce structural ambiguity. It is applied on LAX output.

The sentence generation tool [Ikeda *et al.* 88] generates a Japanese sentence from a internal representation of PST form:

```
{relation/
   {word/ tayo-ru}
 role/
   {goal/
      {comp/
         {modificand/
            {word/ megumi},
   ...
 modal/
   {mood/ [inevitable]}}.
```

CIL is written in ESP, while other two tools were transplanted to CESP.

3.6 Software Publication

Software tools introduced above will be distributed in source codes from ICOT. Programs written in CESP can be executed on several UNIX workstations. Access AIR (AI Language Research Institute, Ltd.) for detail information of CESP language and how to obtain it.

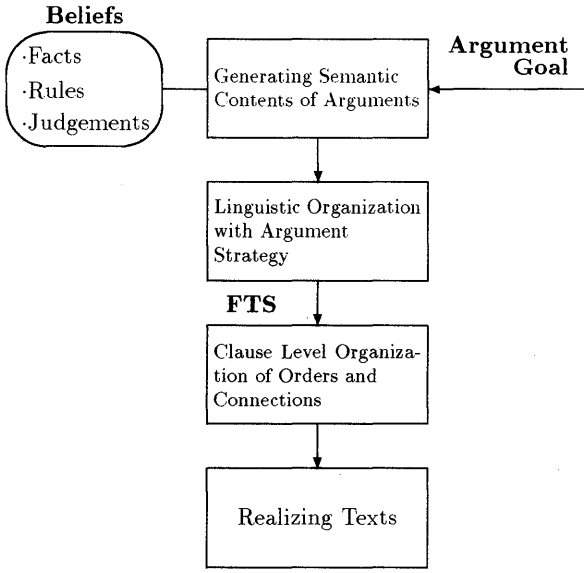


Figure 7: Configuration of the Argument Text Generation System

4 Discourse Processing Systems

In the experiments of discourse processing systems, we have collected rules for language phnomena concerning discourse, and developed several experimental systems in this field including text generation, discourse structure construction, and dialog systems.

The text generation system has a system's belief as a knowledge-base, and produces one or more paragraphs of text concerning to a given theme based on its belief and judgement using rhetorical heuristics.

The discourse structure construction system uses rules for classification of sentence types and of relationship between sentences in a discourse to construct a tree-like discourse structure of a given text.

4.1 Argument Text Generation

As described in the previous section, we have developed sentence generation tool as one of the LTB tools. This program generates single sentence from an internal representation which specifies many semantic and surface attributes of the sentence precisely [Ikeda *et al.* 88]. As a tool, it is not so convenient because the user must be aware of internal representation and grammatical rules.

Moreover, main topic of sentence generation has shifted to paragraph or full text generation. And the quality of generated sentences has raised higher so that speaker's intention and position can be expressed [Tokunaga and Inui 91]. In order to realize such functions in generation, planning text structure, semantic contents, hearer's intention is important [Appelt 88] [Hovy 85] [Hovy 90a].

Against this background, we developed a generation system for argument text. This system generates a text by which the system tries to pursuade the hearer in a given argument. The configuration of the system is shown in Figure 7. Detailed description of this system is given in the paper [Ikeda *et al.* 92] in ICOT session of this conference.

The system has his belief as a knowledge-base. It contains facts, rules and his judgement about world events. If this judgement is substituted by another, remaining facts and rules left unchanged, then the system draw a different conclusion for the same object.

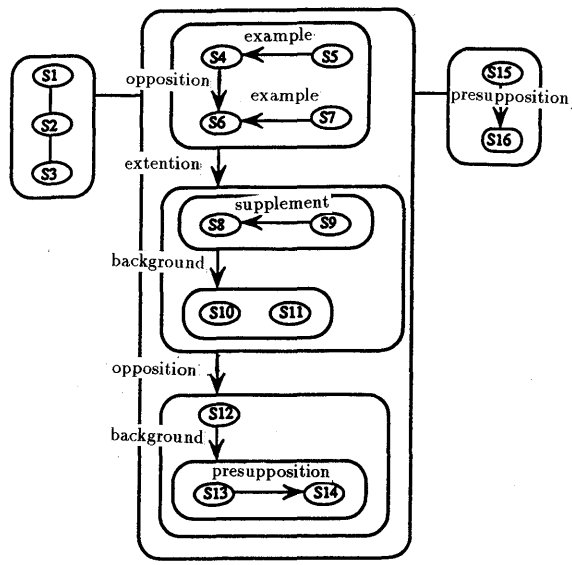


Figure 8: An Example of a Text Structure

4.2 Discourse Structure Extraction

First step of discourse structure extraction is to classify sentences in a context into several sentence types, such as assertive, descriptive, interrogative, and *etc.* Then, using these sentence types and relation between adjacent sentences, sentences will be gathered together into sentence groups. At the same time, relation between groups will be analyzed. Intergroup relationship contains: example, extention, supplement, opposition, background, presupposition, and *etc.* [Ichikawa 78] [Kinoshita *et al.* 89] These groups can be regarded as paragraphs and paragraph segments [Fukumoto 90] [Shibata *et al.* 90] [Fukumoto and Yasuhara 91] [Saitoh *et al.* 91] [Tanaka *et al.* 91] [Sakuma 88] [Tsujii 89] [Yamanashi 89].

Rules for classifying sentence types and those of analyzing intergroup relationship are described in a formal language, and will be published as a "context grammar."

Figure 8 is an example of a text structure of an editorial of Japanese newspaper with 16 sentences.

The experimental system on the Multi-PSI machine will be demonstrated in this conference.

5 Parallel NLP Experiment

As parallel NLP experiment, we have developed a small system for cooperative processing in which morphological analysis, syntax analysis, and semantic analysis are amalgamated into a uniform process in a type inference framework.

Most of the conventional NLP systems have been designed a collection of independently acting modules. Processing in each module is hidden from the outer world, and we use these modules as black-boxes. But since parallel cooperative processing needs internal information being exchanged between modules, we must adopt other framework for parallel NLP.

One answer to this problem is to abstract processing mechanism to merge all such processing as morphology, syntax, semantics, and *etc.* Constraint transformation proposed by Hasida [Hashida 91] is one of the candidates of this framework. We proposed a type inference method [Martin-Löf 84] as another candidates. This type inference mechanism is based on a typed record structure [Sells 85] or a record structure of types similar to ψ-term [Aït-Kaci and Nasr 86], sorted feature structure [Smolka 88], $\mathcal{QUIXOTE}$ [Yasukawa and Yokota 90], order-sorted logic [Schmidt-Schauss 89].

Morphological analysis and syntax analysis is performed by layered stream method [Matsumoto 86]. Roles of process and communication are exchanged in comparison with the method used in PAX [Satoh 90].

This system, running on multi-PSI machine, using a Japanese dictionary with 10,000 nouns, 1000 verbs, 700 concepts, and a Japanese grammar LUG [Sano 91] [Sano and Fukumoto 92], achieves about 12 speed-up rate using 32 processing elements.

Figure 9 shows the relation between number of processors ($1 \sim 32$) and processing time in milli second for a 25-word long sentence.

Figure 10 shows the relation between reductions and speed-up ratio for various evaluation sentences.

The detail of this system will be presented in the paper [Yamasaki 92] submitted to this conference.

Acknowledgment

We wish to thank Dr. Kazuhiro Fuchi, director of ICOT Research Center, who gave us a chance to research natural language processing, and also Dr. Shunichi Uchida, Manager of Research Division, for his helpful advise on the fundamental organization and direction of our research.

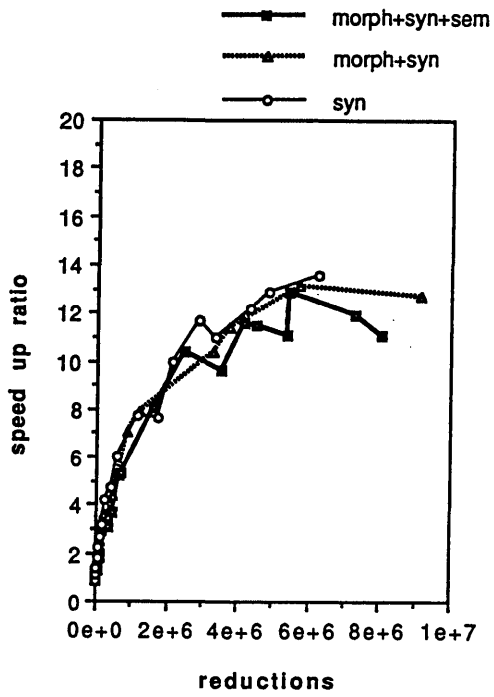


Figure 9: Performance of Experimental System (1)

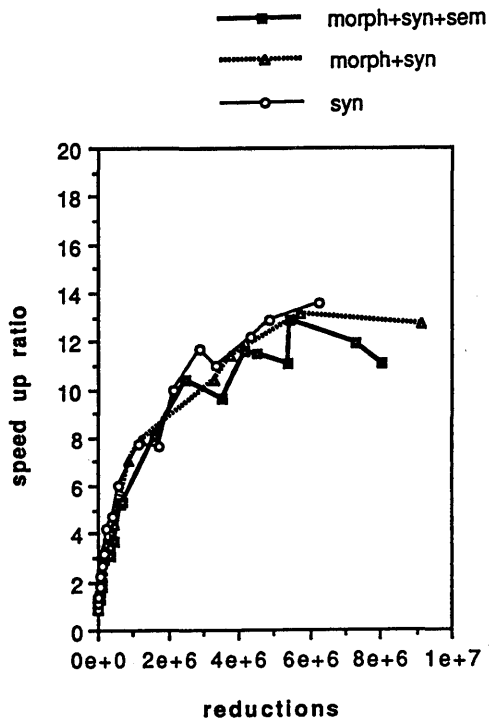


Figure 10: Performance of Experimental System (2)

References

[Abe *et al.* 91] H. Abe, T. Okunishi, H. Miyoshi, and Y. Obuchi. A Sentence Division Method using Connectives. In *Proc. of the 42nd Conference of Information Processing Society of Japan* (in *Japanese*). 1991. pp. 13-15.

[Aït-Kaci and Nasr 86] H. Aït-Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance, *The Journal of Logic Programming*, Vol. 3, No. 3, Oct. 1986.

[Aizawa and Ehara 73] T. Aizawa and T. Ehara. Kana-Kanji Conversion by Computer (in *Japanese*), *NHK Technical Research*, Vol. 25, No. 5, 1973.

[Appelt 85a] D. E. Appelt. *Planning English Sentences*, Cambridge University Press, 1985.

[Appelt 85b] D. E. Appelt. Bidirectional Grammar and the Design of Natural Language Generation Systems, In *Proc. TINLAP-85*, 1985.

[Appelt 87] D. E. Appelt. A Computational Model of Referring, In *Proc. IJCAI-87*, 1987.

[Appelt 88] D. E. Appelt. Planning Natural Language Referring Expressions. In David D. McDonald and Leonard Bolc (eds.), *Natural Language Generation Systems*. Springer-Verlag, 1988.

[Barwise and Perry 83] J. Barwise and J. Perry. *Situation and Attitudes*, MIT Press, 1983.

[Brooks 86] R. A. Brooks. A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, Vol. Ra-2, No. 1. March, 1986.

[Calder 89] Jonathan Calder, Ewan Klein, Henk Zeevat. Unification Categorial Grammar. In *Proc. of the Fourth Conference of the European Chapter of the ACL*, Manchester, 1989.

[Carlson 89] Lauri Carlson. RUG: Regular Unification Grammar. In *Proc. of the Fourth Conference of the European Chapter of the ACL*, Manchester, 1989.

[Danlos 84] L. Danlos. Conceptual and Linguistic Decisions in Generation. In *Proc. of the International Conference on Computational Linguistics*, 1984.

[De Smedt 90] K. J. M. J. De Smedt. Incremental Sentence Generation. *NICI Technical Report*, 90-01, 1990.

[Fujisaki 89] H. Fujisaki. Analysis of Intonation and its Modelling in Japanese Language. *Japanese Language and Education of Japanese* (in *Japanese*). Meiji Shoin Publishing Co., 1989, pp. 266-297.

[Fukumoto and Sano 90] F. Fukumoto, H. Sano. Restricted Dependency Grammar and its Representation. In *Proc. The 41st Conference of Information Processing Society of Japan* (in *Japanese*), 1990.

[Fukumoto 90] J. Fukumoto. Context Structure Extraction of Japanese Text based on Writer's Assertion. In *Research Report of SIG-NL*, Information Processing Society of Japan (in *Japanese*). 78-15, 1990.

[Fukumoto and Yasuhara 91] J. Fukumoto and H. Yasuhara. Structural Analysis of Japanese Text. In *Research Report of SIG-NL*, Information Processing Society of Japan (in *Japanese*). 85-11, 1991.

[Grosz and Sidner 85] B. Grosz and C. L. Sidner. The structures of Discource Structure, Technical Report CSLI, CSLI-85-39, 1985.

[Hashida 91] K. Hasida. Aspects of Integration in Natural Language Processing, *Computer Software*, Japan Society for Software Science and Technology, Vol. 8, No. 6, Nov. 1991.

[Hovy 85] E. H. Hovy. Integrating Text Planning and Production in Generation. In *the Proceedings of the International Joint Conference on Artificial Intelligence*. 1985.

[Hovy 87] E. H. Hovy. Interpretation in Generation. In *the Proceedings of 6th AAAI Conference*. 1987.

[Hovy 88] E. H. Hovy. *Generating Natural Language under Pragmatic Constraints*. Lawrence Erlbaum Associates, Publishers, 1988.

[Hovy 90a] E. H. Hovy. Unresolved Issues in Paragraph Planning. In *Current Research in Natural Language Generation*. Academic Press, 1990.

[Hovy 90b] E. H. Hovy. Pragmatics and Natural Language Generation. *Artificial Intelligence 43*, 1990. pp. 153–197.

[Ichikawa 78] T. Ichikawa. *An Introduction to Japanese Syntax for Teachers*. Kyoiku Shuppan Publishing Co., 1978.

[Ikeda *et al.* 88] T. Ikeda, K. Hatano, H. Fukushima and S. Shigenaga. Generation Method in the Sentence Generator of Language Tool Box (LTB). In *Proc. of the 5th Conference of Japan Society for Software Science and Technology* (in *Japanese*), 1988.

[Ikeda 91] T. Ikeda. Natural Language Generation System based on the Hierarchy of Semantic Representation (in *Japanese*). *Computer Software*, Japan Society for Software Science and Technology, Vol. 8, No. 6, Nov. 1991.

[Ikeda *et al.* 92] T. Ikeda, A. Kotani, K. Hagiwara, Y. Kubo. Argument Text Generation System (Dulcinea). In *Proc. of FGCS '92*, ICOT, Jun. 1992.

[Katoh and Fukuchi 89] Y. Katoh and T. Fukuchi. *Tense, Aspect and Mood* (in *Japanese*). Japanese Example Sentences and Problems for Foreigners 15. Aratake Publishing Co., Tokyo. 1989.

[Kempen and Hoenkamp 87] G. Kempen and E. Hoenkamp. *An Incremental Procedural Grammar for Sentence Formulation*, Cognitive Science, Vol. 11. 1987.

[Kinoshita 81] S. Kinoshita. Writing Techniques in Scientific Field (in *Japanese*). Chuo-Kouron Publishing Co., 1981. pp. 82-88.

[Kinoshita *et al.* 89] S. Kinoshita, K. Ono, T. Ukita and M. Amano. Discourse Structure Extraction in Japanese Text Understanding. In *Symposium on Discourse Understanding Model and its Application* (in *Japanese*), Information Processing Society of Japan, 1989. pp. 125-136.

[Kodama 87] T. Kodama. Research on Dependency Grammar (in *Japanese*). Kenkyu-sha, 1987. pp. 161-194.

[Kubo *et al.* 88] Y. Kubo, M. Yoshizumi, H. Sano. K. Akasaka and R. Sugimura. Development Environment of the Morphological Analyzer LAX. In *Proc. of the 37th Conference of Information Processing Society of Japan* (in *Japanese*). 1988. pp. 1078-1079.

[Kubo 89] Y. Kubo. Composition of Word Semantics in Morphological Analyzer LAX. In *Proc. of the 39th Conference of Information Processing Society of Japan* (in *Japanese*). 1989. pp. 598-599.

[Kuno and Shibatani 89] S. Kuno, K. Shibatani. New Development in Japanese Linguistics (in *Japanese*). Kuroshio Publishing Co., Tokyo, 1989.

[Littman and Allen 87] D. J. Littman and J. F. Allen. A Plan Recognition Model for Subdialogues in Conversation, *Cognitive Science* 11, 1987. pp. 163-200.

[Mann and Thompson 86] W. C. Mann and S. A. Thompson. Rhetorical Structure Theory: Description and Construction of Text Structure, In *Proc. of the Third International Workshop on Text Generation*, 1986. In Dordrecht (ed.), *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics.* Martinus Nijhoff Publishers, 1987.

[Martin-Löf 84] P. Martin-Löf. Intuitionistic Type Theory — Studies in Proof Thoery, *Lecture Notes*, 1984.

[Masuoka 89] T. Masuoka, Y. Takubo. Basic Japanese Grammar (in *Japanese*). Kuroshio Publishing Co., Tokyo. 1989.

[Matsumoto *et al.* 83a] Y. Matsumoto, M. Seino, H. Tanaka. BUP Translator (in *Japanese*). Bulletin of the Electrotechnical Laboratory, Vol. 47, No. 8, 1983.

[Matsumoto *et al.* 83b] Yuji Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa, BUP: A Bottom-up Parser Embedded in Prolog, *New Generation Computing*, Vol. 1, 1983.

[Matsumoto 86] Y. Matsumoto. A Parallel Parsing System for Natural Language Analysis, *Proc. of 3rd Internatioanl Conference on Logic Programming*, London, 1986. *Lecture Notes in Computer Science* 225, pp. 396-409, 1986.

[Matsumoto and Sugimura 87] Y. Matsumoto and R. Sugimura. A Parsing System based on Logic Programming. In *Proceedings of the International Joint Conference of Artificial Intelligence*, 1987.

[Matsumoto 90] Y. Matsumoto and A. Okumura. Programming Searching Problems in Parallel Logic Programming Languages — An Extention of Layered Streams —. In *Proc. of the KL1 Programming Workshop '90* (in *Japanese*). 1990.

[Maruyama and Suzuki 91] T. Maruyama and H. Suzuki. Cooperative Sentence Generation in Japanese Dialog based on Simple Principles (in *Japanese*). In *Proc. of the 8th Conference of Nihon Ninchi Kagaku Kai* (in *Japanese*). 1991.

[McKeown 85a] K. R. McKeown. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Cambridge University Press, 1985.

[McKeown 85b] K. R. McKeown. Discourse Strategies for Generating Natural-Language Text, *Artificial Intelligence 27*, 1985. pp. 1-41.

[Meteer 90] M. W. Meteer. The 'Generation Gap' — the Problem of Expressibility in Text Planning. *Technical Report.* BBN Systems and Technologies Corporation. 1990.

[Minami 74] F. Minami. The Structure of Contemporary Japanese Language (in *Japanese*). Taishu-kan Publishing Co.. 1974.

[Moens 89] Marc Moens, Jonathan Calder, Ewan Klein, Mike Reape, Henk Zeevat. Expressing Generalizations in Unification-based Grammar Formalisms. In *Proc. of the Fourth Conference of the European Chapter of the ACL*, Manchester, 1989.

[Morioka 87] K. Morioka. *Vocabulary Construction* (in *Japanese*). Meiji Shoin Publishing Co., 1987.

[Morita 89] Y. Morita. *Dictionary of Basic Japanese* (in *Japanese*). Kadokawa Publishing Co., 1989.

[Morita and Matsuki 89] Y. Morita and Y. Matsuki. *Sentence Types of Japanese* (in *Japanese*). ALK Publishing Co., Tokyo. 1989.

[Nagano 86] K. Nagano. *Japanese Syntax — a Grammatical Study* (in *Japanese*). Asakura Publishing Co., 1986.

[Nakajima and Sugimura 89] A. Nakajima and R. Sugimura. Japanese Morphological Analysis with TRIE Dictionary and Graph Stack. In *Proc. of the 39th Conference of Information Processing Society of Japan* (in *Japanese*). 1989. pp. 589-590.

[Nitta and Masuoka 89] Y. Nitta and T. Masuoka (eds.), *Modality in Japanese* (in *Japanese*). Kuroshio Publishing Co., Tokyo. 1989.

[NLRI 81] National Language Research Institute. *Demonstratives in Japanese* (in *Japanese*). Ministry of Finance. 1981.

[NLRI 82] National Language Research Institute. *Particles and Auxiliary Verbs of Japanese* (in *Japanese*). Shuei Publishing Co., Tokyo. 1982.

[NLRI 85] National Language Research Institute. *Aspect and Tense of Contemporary Japanese* (in *Japanese*). Shuei Publishing Co., Tokyo. 1985.

[NLRI 89] National Language Research Institute. *Research and Education of Discourse* (in *Japanese*). Ministry of Finance. 1989.

[Nobukuni 89] Y. Nobukuni. Division Algorithm of Long Sentence, In *Proc. of the 39th Conference of Information Processing Society of Japan* (in *Japanese*). 1989. p. 593.

[Okumura and Matsumoto 87a] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proc. of the 1987 International Symposium on Logic Programming*. San Francisco, September 1987. pp. 224-232.

[Okumura and Matsumoto 87b] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proc. of the Logic Programming Conference '87* (in *Japanese*), 1987. pp. 223-232.

[Pereira 80] Fernando C. N. Pereira, David H. D. Warren. Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence*. Vol. 13, No. 3. 1980. pp. 231-278.

[Saitoh *et al.* 91] Y. Saitoh, M. Shibata and J. Fukumoto. Analysis of Relationship of Adjoining Sentences for Context Structure Extraction. In *Proc. of the 43rd Conference of Information Processing Society of Japan* (in *Japanese*). 1991.

[Sakuma 88] M. Sakuma. Context and Paragraph. *Japanese Linguistics* (in *Japanese*). Vol. 7, No. 2. 1988. pp. 27-40.

[Sano *et al.* 88] H. Sano, K. Akasaka, Y. Kubo and R. Sugimura. Morphological Analysis based on Word Formation. In *Proc. of the 36th Conference of Information Processing Society of Japan* (in *Japanese*), 1988.

[Sano 89] H. Sano. Hierarchical Analysis of Predicate using Contextual Information. In *Symposium on Discourse Understanding Model and its Application* (in *Japanese*), Information Processing Society of Japan, 1989.

[Sano *et al.* 90] H. Sano, F. Fukumoto, Y. Tanaka. Explanatory Description based Grammar — SFTB (in *Japanese*), ICOT-Technical Memo, TM-0885, 1990.

[Sano and Fukumoto 90] H. Sano, F. Fukumoto. Localized Unification Grammar and its Representation. In *Proc. of the 41st Conference of Information Processing Society of Japan* (in *Japanese*), 1990.

[Sano 91] H. Sano. User's Guide to SFTB (in *Japanese*), ICOT, Sep. 1991.

[Sano and Fukumoto 92] H. Sano, F. Fukumoto. On a Grammar Formalism, Knowledge Bases and Tools for Natural Language Processing in Logic Programming. In *Proc. of FGCS '92*, ICOT, Jun. 1992.

[Satoh 90] H. Satoh. Improvement of Parallel Syntax Analyzer PAX. In *Proc. of KL1 Programming Workshop '90* (in *Japanese*), ICOT, Tokyo, 1990.

[Schmidt-Schauss 89] M. Schmidt-Schauß. Computational Aspects of an Order-Sorted Logic with Term Declarations, *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1989.

[Searl 69] J. R. Searl. *An Essay in the Philosophy of Language*, Cambridge University Press, 1969.

[Sells 85] P. Sells. Lectures on Contemporary Syntactic Theories, *CSLI Lecture Notes*, No. 3, 1985.

[Shibata *et al.* 90] M. Shibata, Y. Tanaka and J. Fukumoto. Anaphora Phenomena in Newspaper Editorials. In *Proc. of the 40th Conference of Information Processing Society of Japan* (in *Japanese*), 1990.

[Shinnou and Suzuki 91] H. Shinnou and H. Suzuki. Utilization of Sound Information in Incremental Analysis. In *Research Report of SIG-NL*, Information Processing Society of Japan (in *Japanese*). 85-7. 1991.

[Shiraishi *et al.* 90] T. Shiraishi. Y. Kubo and M. Yoshizumi. Format of Morpheme Dictionary and Dictionary Improvement. In *Proc. of the 41st Conference of Information Processing Society of Japan* (in *Japanese*), 1990. pp. 193-194.

[Smolka 88] G. Smolka. A Feature Logic with Subsorts, IBM Deutschland, Stuttgart, Germany, *LILOG Report*, No. 33, May 1988.

[Sugimura *et al.* 88] R. Sugimura, K. Akasaka, Y. Kubo, Y. Matsumoto and H. Sano. LAX — Morphological Analyzer in Logic Programming. In *Proc. of the Logic Programming Conference '88* (in *Japanese*), 1988. pp. 213-222.

[Sugimura and Fukumoto 89] R. Sugimura. F. Fukumoto. Dependency Analysis by Logic Grammar. In *Symposium on Discourse Understanding Model and its Application* (in *Japanese*). Information Processing Society of Japan. 1989.

[Suzuki and Tsuchiya 90] H. Suzuki and S. Tsuchiya. Incremental Interpretation of Japanese Utterance. In *Proc. of the 7th Conference of Nihon Ninchi Kagaku Kai* (in *Japanese*). 1990. pp. 46-47.

[Tanaka *et al.* 91] Y. Tanaka, M. Shibata and J. Fukumoto. Repetitive Occurrence Analysis of a Word in Context Structure Analysis System. In *Proc. of the 43rd Conference of Information Processing Society of Japan* (in *Japanese*), 1991.

[Teramura *et al.* 87] H. Teramura, Y. Suzuki, N. Noda and M. Yazawa. *Case Study in Japanese Grammar* (in *Japanese*). Outousha Publishing Co., Tokyo. 1987.

[Tokunaga and Inui 91] T. Tokunaga and K. Inui. Survey of Natural Language Sentence Generation in 1980's. In *Journal of Japanese Society for Artificial Intelligence* (in *Japanese*). Vol. 6, Nos. 3–5, 1991.

[Tomita 87] M. Tomita. An Efficient Augmented Context Free Parsing Algorithm. *Computational Linguistics* 13, 1-2, 31-46, 1987.

[Tsujii 89] J. Tsujii. Context Processing. In *Symposium on Natural Language Processing* (in *Japanese*), Information Processing Society of Japan. 1988. pp. 75-87.

[Ueda and Chikayama 90] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*. Vol. 33, No. 6, Dec. 1990. pp. 494-500.

[Yamanashi 86] M. Yamanashi. *Speech Act* (in *Japanese*). Taishukan Publishting Co., 1986.

[Yamanashi 89] M. Yamanashi. Discourse, Context and Inference. In *Symposium on Discourse Understanding Model and its Application* (in *Japanese*). Information Processing Society of Japan. 1989. pp. 1-12.

[Yamasaki 92] S. Yamasaki. A Parallel Cooperative Natural Language Processing System — Laputa. In *Proc. of FGCS '92*. ICOT. Jun. 1992.

[Yasukawa and Yokota 90] H. Yasukawa and K. Yokota. The Overview of a Knowledge Representation Language QUIXOTE, ICOT (draft), Oct. 21, 1990.

[Yoneda *et al.* 89] J. Yoneda, Y. Kubo, T. Shiraishi and M. Yoshizumi. Interpreter and Debugging Environment of LAX. In *Proc. of the 39th Conference of Information Processing Society of Japan* (in *Japanese*), 1989. pp. 596-597.

[Yoshida and Hidaka 87] M. Yoshida and S. Hidaka. *Studies on Documentation in Standard Japanese* (in *Japanese*). 1987.

[Yoshimura *et al.* 82] K. Yoshimura, T. Hidaka and M. Yoshida. On Longest Maching Method and Word Minimizing Method in Japanese Morphological Analysis. In *Research Report of SIG-NL*. Information Processing Society of Japan (in *Japanese*). 30-7, 1982.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Experimental Parallel Inference Software

Katsumi Nitta Kazuo Taki Nobuyuki Ichiyoshi

Seventh Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{nitta,taki,ichiyoshi}@icot.or.jp

Abstract

As tools to develop large scale intelligent systems, ICOT has developed parallel inference machines PIMs, a parallel logic programming language KL1 and an operating system PIMOS. In order to evaluate the appropriateness of these tools to the development of practical intelligent systems, we have developed four application programs in KL1 — the LSI-CAD system, the Genome Analysis System, the Legal Reasoning System and the Go Playing Game System —, and cooperating manufacturers have developed eight application programs. They cover a wide range of knowledge processing techniques such as case-based reasoning, model-based reasoning, qualitative reasoning and machine learning.

To obtain high performance from each application program, we have developed parallel programming techniques such as concurrent algorithms and load balancing. Moreover, we analyzed the performance of parallel programming technology theoretically. The result forms good guidelines for the selection of parallel programming techniques.

We introduce each application program and the results of performance analysis, and discuss our experiences of parallel programming.

1 Introduction

As tools to develop knowledge processing systems, ICOT has developed an experimental parallel inference machine Multi-PSI and five models of parallel inference machine PIMs [Uchida *et al.* 1988] [Goto *et al.* 1988]. They are MIMD machines on which user's programs written in the parallel logic programming language KL1 can run in parallel [Chikayama 1992]. As KL1 is based on the theory of first order predicate logic, it is useful to represent human knowledge naturally and to formalize inference processes naturally. Therefore, we can develop large-scale intelligent systems easier by using PIMs and KL1. However, if we develop KL1 programs naively, we cannot obtain high performance because the performance can be affected by sequential bottlenecks and various parallelization overheads; Good parallel algorithms and load distribution techniques have to be developed. Moreover, to develop efficient parallel programs, we have to understand the characteristics of the KL1 language and the architectures of Multi-PSI and PIMs (Figure 1). As these parallel programming technologies are closely related, when we develop KL1 programs, we have to choose suitable techniques carefully. Therefore, we need guidelines for selecting suitable parallel programming techniques and for estimating the relation between data size, number of processors, and performance. To get such guidelines, in addition to developing application programs, we have to conduct theoretical analysis of parallel programming techniques.

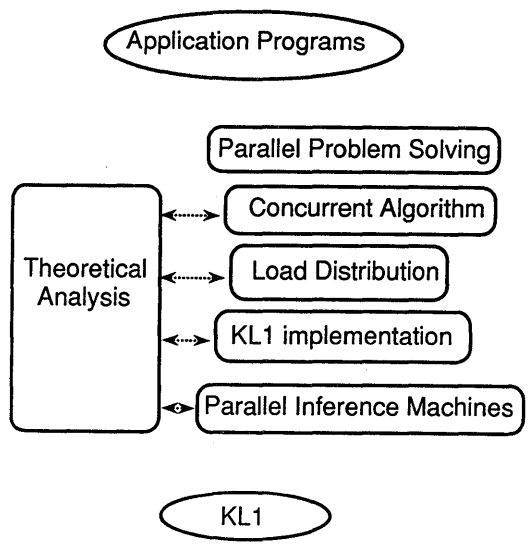


Figure 1: Parallel programming technologies

We have developed parallel application programs to achieve the following goals.

- Evaluation of applicability of PIMs to developing practical intelligent systems:

As PIMs solve problems efficiently by parallel inference, developing large scale systems using them is easier than using other computers. We wish to cultivate application fields and develop AI techniques where PIMs are effectively used.

- Development of Parallel Programming Techniques:
  By analyzing the behavior of application programs, we can extract parallel programming techniques to obtain high performance. A library of these techniques will help to develop new parallel programs.

In Section 2, we will give an overview of the research activities of the seventh research laboratory of ICOT. Section 3 presents application programs developed inside ICOT, and Section 4 presents application programs developed outside ICOT. In Section 5, the research activity in performance analysis is reported. In Section 6, we summarize the experiences of parallel program development.

2 Research Activities

As we explained in the previous section, to develop intelligent systems on PIMs, we have to cover a wide range of technologies from the knowledge of human experts to the features of hardware. To manage the various researches effectively, we organized the researchers of the seventh research laboratory into four *Application Groups* and one *Performance Analysis Group* (Figure 2). The roles of the *Application Groups* and the *Performance Analysis Group* are to develop specific application programs, and to give guidelines on parallel programming techniques by analyzing the behavior of KL1 programs theoretically.

Following are the researches of the Application Groups. To acquire knowledge from human experts effectively, these groups established four working groups: parallel IC CAD (PIC), genetic information processing (GIP), advanced design system (ADS), and knowledge architecture (KAR).

- LSI-CAD System:
  The LSI design process consists of several stages, such as architecture design, function design, logic design, micro program design, logic simulation and layout design. This group has developed the following two systems.
  - Logic Simulator
  - LSI Layout Systems
- Genome Analysis System:
  One of the most important targets of genome analysis is to interpret the meanings of protein sequences. This group has developed the following systems.
  - Protein Sequence Analysis System
  - Protein Folding Simulation Program
  - Protein Structure Analysis Program
- Legal Reasoning System:
  The difficulty of legal reasoning stems from the ambiguity of legal concepts. To deal with ambiguous concepts, this group has developed a legal reasoning system with a rule-based engine and a case-based engine.
- Go Playing Game System:
  The game of *go* is a traditional Japanese board game. This group has developed a parallel *go* playing game system.

In the next section, we will present an overview of each system.

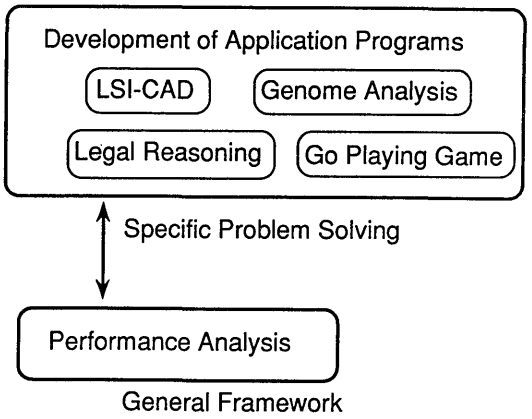


Figure 2: Research groups

Besides the above application programs, cooperating manufacturers have developed knowledge processing systems in order to evaluate the appropriateness of PIMs to these fields.

- Co-HLEX: Co-operative Recursive LSI Layout Problem Solver
  (hierarchical and cooperative problem solving)
- Cooperative Logic design Expert System on a Multi-Processor
  (assumption-based reasoning, cooperative problem solving)
- Case-based circuit design system
  (case-based reasoning)
- High Level Synthesis by Parallel Rule-based Annealing
  (rule-based annealing)
- Design Supporting System based on Deep Reasoning
  (qualitative reasoning)

- A Diagnostic and Control Expert System Based on a Plant Model
  (model-based reasoning, qualitative reasoning)

- Adaptive Model-based Diagnostic System (model-based reasoning, machine learning)

- Motif Extraction System
  (genetic algorithm, machine learning)

These systems cover various knowledge processing systems such as CAD systems, diagnosis systems, and control systems. They are related to various AI techniques such as case-based reasoning, qualitative reasoning, model based reasoning, and machine learning.

We will introduce these systems in Section 4.

3 Overview of Application Programs (1)

3.1 Logic Simulator

3.1.1 Background

A logic simulator is used to verify not only the functions of designed circuits but also the timing of signal propagation. Since logic simulation is one of the most time-consuming stages in LSI design, faster simulators are urgently needed. A parallel logic simulator is one likely way of producing quick simulation.

Parallel logic simulation is treated as a typical application of parallel discrete event simulation (PDES). PDES can be modeled so that several objects (state automata) change their states by communicating with each other. A message has the information of an event whose occurrence time is stamped on the message (time-stamp). Since messages should be received and evaluated in the time-stamp order by their destination objects, the time-keeping mechanism is essential for efficient execution of PDES. Several mechanisms have been proposed for PDES time-keeping, however, each has its own peculiar shortcomings.

We are targeting an efficient logic simulator on PIM, which is a distributed memory MIMD machine. We adopted the Time Warp mechanism (TW), which has been considered to contain a heavy overhead — a rollback process. In practice, however, TW has never been evaluated in detail on MIMD machines. We expected that TW would be a suitable logic simulator on large-scale MIMD machines with some devices that reduced the rollback overhead. Thus, a local message scheduler, an antimessage reduction mechanism, and a load distribution scheme were added to our system and evaluated.

3.1.2 Overview of Logic Simulator

The system simulates combinatorial circuits and sequential circuits that have feedback loops. It handles three values: Hi, Lo, and X (unknown). A different delay time can be assigned to each gate (non-unit delay model). Since this simulator handles gates only, flip-flops and other functional blocks should be completely decomposed into gates.

The Time Warp mechanism (TW) [Jefferson 1985] was proposed by D. R. Jefferson. In PDES using TW, each object usually acts according to received messages and also records the history of messages and states, assuming that messages arrive chronologically. But when a message arrives at an object out of time-stamp order, the object rewinds its history (this process is called rollback), and makes adjustments as if the message had arrived in correct time-stamp order. After rollback, ordinary computation is resumed. If there are messages which should not have been sent, the object also sends antimessages in order to cancel those messages.

Since TW contains its own peculiar overheads caused by the rollback processes, some device for reducing the overheads is needed for quick simulation. Furthermore, inter-PE communication overheads must be reduced because the simulator works on a distributed memory machine such as PIM.

For these purposes, a load distribution scheme, a local message scheduler, and an antimessage reduction mechanism are included in our simulator. These are expected to reduce the overheads described above and might promote the efficient execution of the simulator.

Each device is outlined below.

- Cascading-Oriented Partitioning

We propose the Cascading-Oriented Partitioning strategy for partitioning circuits to attain high-quality load distribution (Figure 3).

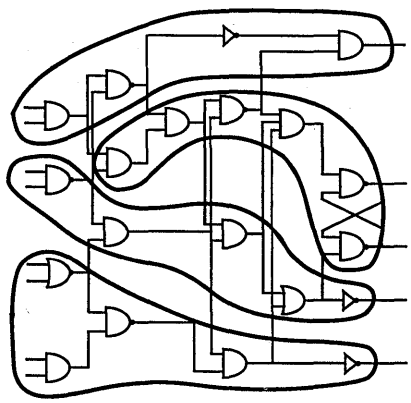


Figure 3: Cascading-Oriented Partitioning

This scheme provides adequate partitioning solutions that satisfy these three requirements: load balancing,

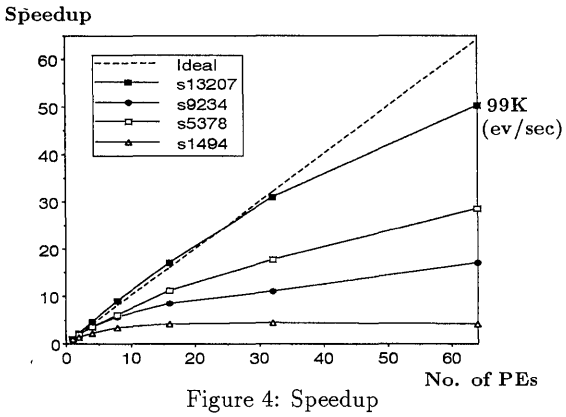


Figure 4: Speedup

keeping inter-PE communication frequency low, and deriving a lot of parallelism.

- Local Message Scheduler

During simulation, there are usually several messages to be evaluated in a PE. When TW is used, the bigger time-stamp a message has, the more likely the message is to be rolled back. For this reason, appropriate message scheduling in each PE is needed for reducing rollback frequency.

- Antimessage Reduction

As long as messages are sent through the KL1 stream, messages arrive at their receiver in the same order as they are transmitted. In this environment, subsequent antimessages can be reduced. We adopted this optimization, expecting that it would reduce the rollback cost.

3.1.3 Result

We executed several experimental simulations on the Multi-PSI. Four sequential circuits, presented in ISCAS'89, were simulated in our experiments.

Figure 4 shows the speedup figures when the circuits were simulated using various numbers of PEs. The best performance is also shown there. In the best case, very good speedup of 48-fold was attained using 64 PEs. Approximately 99K events/sec performance, fairly good for a full-software logic simulator, was also attained. This experiment revealed that the Time Warp mechanism would be an efficient time-keeping mechanism.

In addition, we analyzed several factors which possibly limited speedup. Details are reported in [Matsumoto *et al.* 1992].

3.2 LSI Layout Systems

3.2.1 Background

The LSI layout consists of two stages. The first is *placement*, which determines the physical position of the circuit components. The next is *routing*, which finds the paths between terminals of the circuit components. These are the most time-consuming stages in LSI design. Therefore high performance layout CAD systems lead to a shorter LSI design period.

Our aim is to study concurrent algorithms and load-balancing methodologies through design and development of parallel layout programs. Also, we are targeting the system to attain a high quality layout running on Multi-PSI and PIM.

3.2.2 Overview of LSI Layout System

(1) Placement System Our placement system is implemented for the standard cell type LSI without any macro blocks. The standard cells have uniform height and variant widths. These cells are assigned into multiple cell-blocks so as to minimize the chip area (strictly speaking, the total estimated wire length). The cell placement problem is a combinatorial optimization problem. As a powerful technique to solve such problems, simulated annealing (SA) is well-known. In order to execute SA efficiently, cooling schedules are important. In our placement system, the time-homogeneous parallel SA algorithm [Kimura *et al.* 1991], which constructs appropriate cooling schedules automatically, was adopted. Figure 5 shows an outline of this algorithm.

(2) Routing System Our routing system finds paths based on the look-ahead line search algorithm [Kitazawa 1985]. This algorithm provides high quality solutions in a short execution time, however, it was originally proposed with assumption of sequential execution. We introduced a new programming style based on a concurrent objects model in routing problems, and improved the basic algorithm to make it suitable for parallel execution. The concurrent objects model is expected to derive parallelism of small grain size. We designed the concurrent algorithm so that objects=processes corresponds to

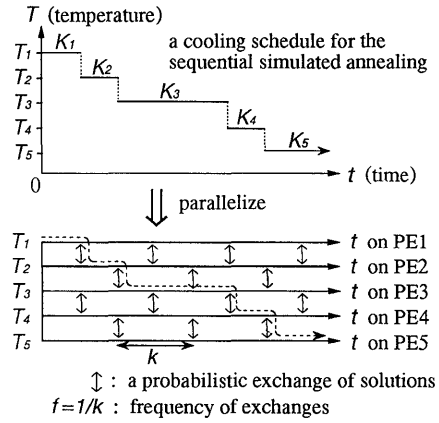


Figure 5: Time-homogeneous parallel simulated annealing

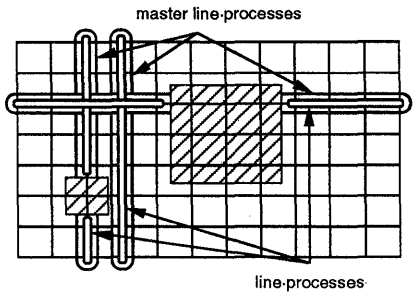


Figure 6: Master line processes and line processes

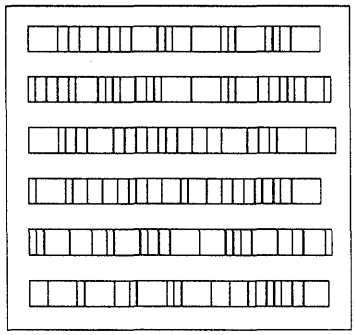


Figure 7: Placement results

every line segment on a routing grid. As in Figure 6, each process corresponds to each grid line (*master line process*) and line segment (*line process*) on it. A master line process manages line processes on the same grid line and passes messages between the line processes and crossing line processes.

3.2.3 Result

(1) Placement System The MCNC benchmark data consisting of 125 cells and 147 nets was chosen for our measurements. In the initial placement, the value of energy was 911520 (the lower bound of the chip area is estimated as $1.372[mm^2]$).

When we executed our program for 30 minutes using 64 processors, the final energy was 424478 (the lower bound of the chip area is estimated as $0.615\ [mm^2]$).

Experimental results showed that final energy is reduced by 56.0 percent in comparison to the initial energy. Figure 7 shows the placement results.

(2) Routing We evaluate our router from the following three points of view using real LSI chip data. (1) Data size vs. Speedup, (2) Parallelism vs. Wiring Rate, (3) Comparison with a general purpose computer.

Figure 8 shows the system performance when the routing program was executed using various numbers of PEs. The size of DATA2 is larger than DATA1. In the best case, 24-fold speedup was attained using 64 PEs.

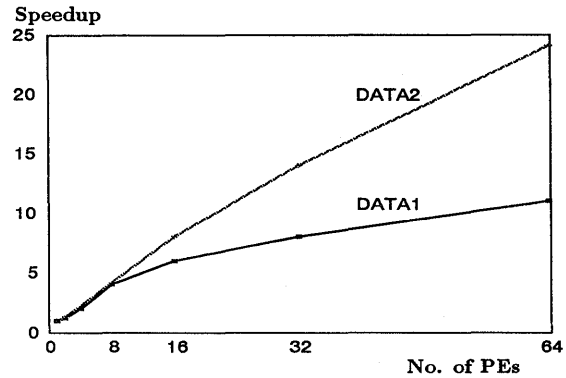


Figure 8: Speedup

Other experimental results are reported in [Date *et al.* 1992].

3.3 Protein Sequence Analysis Programs

3.3.1 Background

A primary structure of protein is a linear chain of amino acids. After a protein is created in the cell, it is folded and forms a complex structure.

The similarity analysis of protein sequences by the use of *multiple alignment* is an important technique for predicting the function and higher order structure of proteins and for drawing phylogenetic trees of creatures. An alignment is realized by lining the sequences with corresponding characters (amino acids) directly above one another as follows.

```
...YICSFADCGAAYNKNWKLQAHLC-KH...
...FPCKEEGCEKGFTSLHHLTRHFL-TH...
...FTCDSDFCDLRFTTKANMKKHFNRFH...
```

Until recently, multiple alignment was produced by hand by biologists. However, with the increasing rate of determination of protein sequences, computer assistance in multiple alignment is becoming indispensable.

It is well-known that once a similarity value between amino acids is given, the multiple alignment problem can be solved theoretically by Dynamic Programming (DP)[Needleman *et al.* 1970]. An alignment algorithm by DP method is the same as finding the shortest path in a network constructed by input sequences (Figure 9). N-way DP can align n sequences simultaneously and can derive the optimal alignment of these sequences.

One problem with DP is the incredible computational time it requires. N-way DP takes computational time in the order of the n-th power of the sequence length. To keep this expansion of computational time manageable, nearly all multiple alignment systems developed so far employ 2-way DP as a base and combine the results of 2-way DP to produce multiple alignment [Barton 1990].

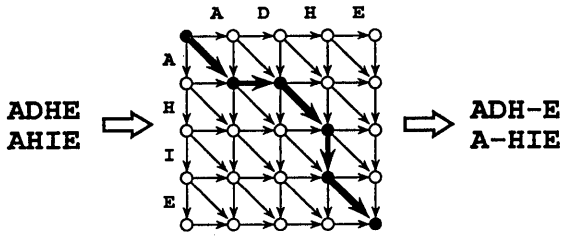


Figure 9: Alignment by 2-way DP

This class of alignment methods is good because of the small computational time required, but this is not sufficient to produce an alignment of sequences when their similarities are low.

3.3.2 Overview of Protein Sequence Analysis Programs

To produce multiple alignments of high-quality with small increases in computational time, we developed several multiple alignment systems. MASCOT (Multiple Alignment System developed by iCOT, see Figure 10) is a multiple alignment system based on DP [Hirosawa *et al.* 1991].

When protein sequences are given to MASCOT, MASCOT, firstly, classifies them into several clusters based on the similarities of sequences. Next, for each cluster, sequences are aligned from the nearest tree sequences using 3-way DP. Then, each intra-cluster alignment is refined by the simulated annealing method (Figure 5). Finally, each intra-cluster alignment is merged into a single alignment.

3.3.3 Result

Each module of MASCOT is described by the KL1 and is executed on the Multi-PSI. Though MASCOT requires more computation than conventional alignment systems due to the use of 3-way DP, parallel execution by the parallel inference machine [Ishikawa *et al.* 1991] can reduce the total time. Figure 11 shows the speedup of 3-way DP versus the number of processors used. 128 processors are about 64 times faster than a single processor.

MASCOT can produce a biologically valuable result. A resultant alignment shows clear consensus patterns in core alignments and discernible patterns in the alignment of each cluster. We think that this is a promising way to compare these kinds of pattern information with known motif information so that integrated information can be useful for attachment-alignment and intra-cluster alignment. We are now investigating how to use knowledge engineering to realize such an extension of MASCOT.

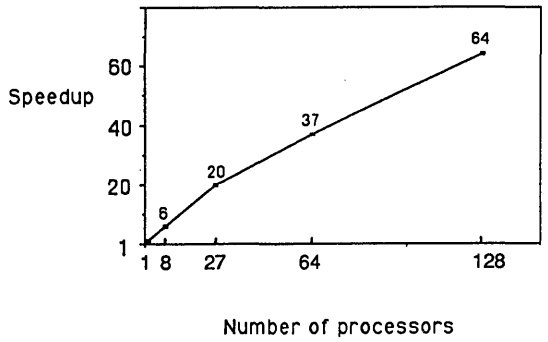


Figure 11: Speedup in 3-way dynamic programming

3.4 Folding Simulation Program

3.4.1 Background

Folding simulation simulates the process of protein formation from its stretched state to its native folded state by computer. This research topic has held the interest of biologists for a quarter of a century because while we can determine the order of amino acids in a sequence of protein extremely easily, it is very difficult to determine the structure of a protein. X-ray crystallography and NMR(Nuclear Magnetic Resonance) can be used to determine structure. However, both require plenty of time from months to a year.

One of the most frequently employed approximation methods is lattice representation [Ueda *et al.* 1978] [Skolnick and Kolinsky 1991], which restricts the position of amino acids in 3-dimensional lattice cells.

3.4.2 Overview of Folding Simulation Program

We applied *time homogeneous parallel (temperature parallel)* simulated annealing (Figure 5) to the folding simulation problem [Hirosawa *et al.* 1992]. Water-counting, which uses lattice representation (Figure 12) and employs only hydrophobic interaction, is introduced to formulate folding simulation as an optimization problem. In lattice cells, any place where protein is not present wiill be filled with water.

The energy to be minimized is expressed in the following formula.

$$E(Energy) = \sum_{m}^{sidechains}(Water\ Count_m - 1) \times Hydrophobicity_m$$

$$Water\ Count_m = \frac{Number\ of\ adjacent\ cells\ (of\ side\ chain)\ occupied\ by\ other\ amino\ acids}{The\ number\ of\ adjacent\ cells\ of\ the\ side\ chain}$$

The energy can be reduced both by increasing the amount of water around the hydrophilic amino acid and by reducing the amount of water around the hydrophobic amino acid. The minimization of energy has the effect of inviting hydrophobic amino acids toward the center

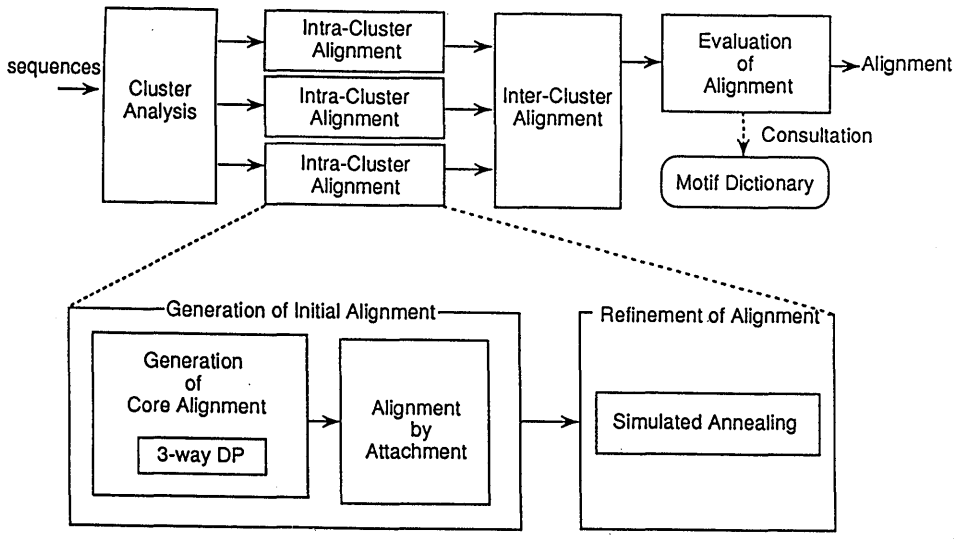


Figure 10: Multiple sequence alignment system: MASCOT

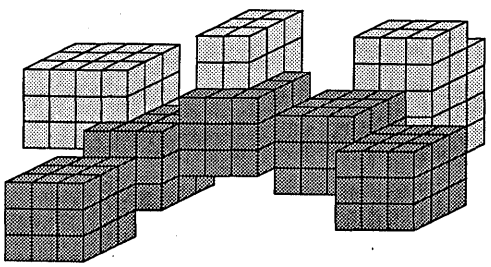


Figure 12: Representation of a section of protein: main chains(shaded) and side chains(unshaded)

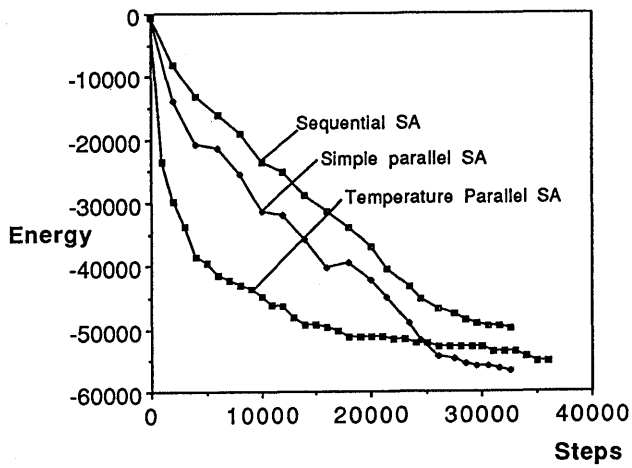


Figure 13: Energy history of folding simulation

of the protein where there is less water and to oust hydrophilic amino acids to the surface of the protein where water is abundant. These effects serve to produce protein that has a similar distribution of hydrophobic amino acids and hydrophilic amino acids within the protein structure.

3.4.3 Result

We selected flavodoxin, whose structure is known, as the protein to be simulated. This protein is of a medium size and has 138 amino acids. We ran the folding simulation program using temperature parallel SA on Multi-PSI using 20 processors over 10 days. This corresponds to 30,000 cycles. We also ran the folding simulation program using *simple parallel* SA in 30,000 cycles, also with 20 processors.

We made the following observations from the energy history of simulation (Figure 13).

1. Two kinds of parallel SAs had better results within a fixed time than sequential SA. This is simply the effect of multiple processors.

2. Up to the middle stage of simulation, temperature parallel SA is always better than simple parallel SA. This is because temperature parallel SA can produce optimal solutions at that time.

3. Two kinds of parallel SAs have almost the same final energy value.

3.5 Protein Structure Analysis Programs

3.5.1 Background

One of the most important problems in the field of structural biology and biophysics is protein structure prediction. Structural biologists have proposed many methods to solve the structure prediction problem. Still, the accuracy of secondary structure prediction (i.e. to know the local feature of a protein structure), which seems to be the easiest part of protein structure prediction, is far below the biological demand.

3.5.2 Overview of Protein structure analysis programs

We plan to solve this difficult problem by a three-phase strategy. In the first phase, we should develop a effective method for representing the structure of protein. Secondly, we are to analyze the statistical relation between the representation and the sequence of a protein, and to obtain a statistic prediction method. Finally, we are planning to analyze which part of the prediction is statistically imprecise by logical consideration in order to know the limits of the statistical prediction method. We also plan to improve the prediction method by using logical knowledge gained from analysis. This plan should ensure that the parallel inference machine is used effectively.

At the moment, we are in the first phase, and have obtained a new way of representing the structure of protein produced by multi-variate analysis (Figure 14). The three dimensional distribution of the amino asid residues which are serial in a protein sequence is easily characterized by each standard deviation on the three main axes of the distribution. This gives us the local coordinates for analyzing the local structure.

3.5.3 Result

As the result, we found it possible to numerically represent the local structure of protein, and we can recognize its secondary structure from this new representation of protein. This numerical representation, which seems to be suitable for numerical operations such as regression analysis, may be quantized into a symbolic representation for logical or symbolic operations (Figure 15).

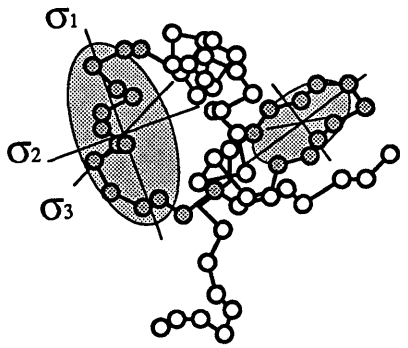


Figure 14: Main axes of the distribution of amino acid residues

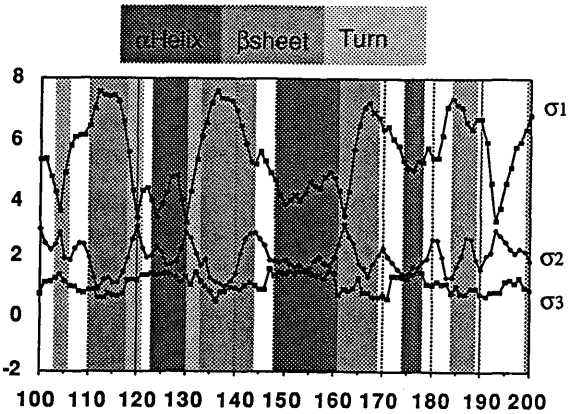


Figure 15: Spatial distribution of amino acids of protein sequence

3.6 A Legal Reasoning System

3.6.1 Background

Legal knowledge consists of statutory laws and old cases. As a statutory law is a set of legal rules, inference by a statutory law is realized as rule-based reasoning. However, legal rules often contain legal predicates (legal concepts). Some legal concepts are ambiguous and their strict meanings are not fixed until the rules are applied to actual facts. To apply legal rules to actual facts, rule interpretation and matching between legal concepts and concrete facts are needed. To realize this, old cases are often referenced and their explanations are reused. Consequently, legal reasoning can be modeled as a mixed paradigm of rule-based reasoning and case-based reasoning.

However, there are some difficulties in developing a practical legal reasoning system. Firstly, as there are many legal rules and many old cases, it takes a long time to search for similar cases and to draw conclusions based on them. Secondly, to manage several inference engines, a complex mechanism to control inference is needed.

To solve these problems by parallel inference, we developed a legal reasoning system, HELIC-II, on the parallel inference machine.

3.6.2 Overview of the Legal Reasoning System

HELIC-II draws legal conclusions for a given case by referencing a statutory law and old cases and outputing them in the form of inference trees [Nitta *et al.* 1992].

HELIC-II consists of a rule-based engine and a case-based engine (Figure 16). The rule-based engine refers to legal rules and draws legal consequences logically. The case-based engine generates legal concepts from given facts by referring to similar old cases.

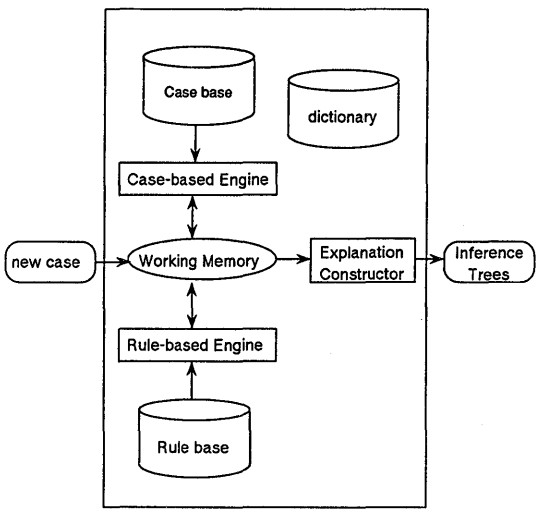


Figure 16: Architecture of HELIC-II

Rule-based inference As there are many legal rules, a fast rule-based engine is needed. Moreover, legal rules sometimes have exceptional rules, the rule-based engine has to be added some mechanism to handle nonmonotonic reasoning.

The rule-based engine of HELIC-II is based on the parallel theorem prover MGTP (Model Generation Theorem Prover) [Fujita *et al.* 1991]. Given a set of non-Horn clauses, MGTP generates models which satisfy all input clauses by parallel inference.

To use MGTP as a rule-based engine of legal rules, and to obtain high performance by pipeline effect, we added several extended functions to the original MGTP.

Case-based inference A judicial precedent (old case) consists of arguments by both sides and the opinion of judges and a final conclusion. We represent an old case as a *situation* and some *case rules*.

A *situation* contains informations on the occurrences of the case and represents a set of events/objects and their *temporal relations*. Arguments by both sides are represented as a set of *case rules*.

The function of the case-based engine is to generate legal concepts by referring to similar old cases. In the first stage, the engine searches for similar cases from the case base. Old cases are distributed to each processor(PE) of the Multi PSI, and similarities between the new case and old cases are evaluated in parallel. In the second stage, similarities between case rules of selected cases and the new case are measured using a Rete-like network (Figure 17), and new arguments are constructed.

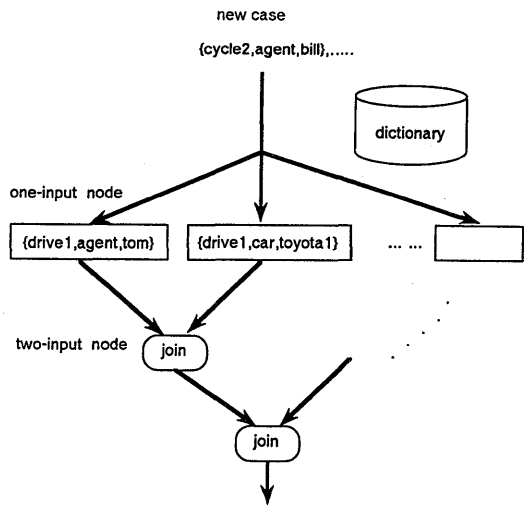


Figure 17: Rete-like network

3.6.3 Results

We observed that HELIC-II can solve several cases of the Penal Code. Figure 18 shows the speedup in the second stage of the case-based engine. We obtained more than 50-fold speedup using the 64PEs of the Multi-PSI.

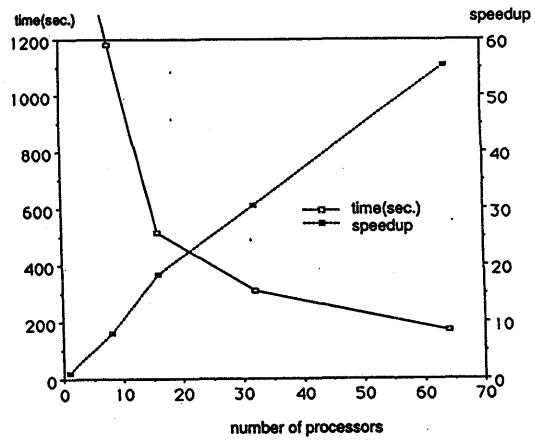


Figure 18: Performance of the case-based engine

3.7 Go Playing Game System "GOG"

3.7.1 Background

Go is a popular board game played traditionally in Japan, China, and Korea. Go is played using black and white stones and a 19 × 19 grid. The two players alternately place black and white stones on the grid intersections. The goal is to gain more secure territories than your opponent. It is a perfect information game.

Go has been a difficult game for computers to play. There have been no go-playing programs that match the ability of average human go-player. The difficulty of constructing a go-playing program comes mainly from the fact that (1) the branching factor of an average game tree is too large for brute force searches to be feasible, and (2) a simple and good board evaluation function does not exist.

As a go-playing program requires basic AI techniques such as searching, processing ambiguous patterns, exceptional processing, and cooperative problem solving, it is a suitable research subject for knowledge processing technologies.

We are trying to build a strong go program using the computing power of the parallel inference machines. We are aiming at the strength of GOG (GO Generation) with the ability of the average human player.

3.7.2 Overview of GOG

GOG has the following three features.

1. It simulates the thinking mechanism of a human player.
2. The large tasks are performed in parallel.
3. The new "flying corps" technique has been applied to improve the strength of GOG considerably while retaining its real-time response.

Simulating the Thinking Mechanism of a Human Player The process in which the GOG system determines its next moves comprises three stages (Figure 20). When the system receives the enemy's move, it first recognizes the board configuration. And then, it generates many candidate moves. It rates those moves and selects the one with the highest value as the next move.

- Board Recognition

  The raw data of the board configuration is simply the state of every board position, which is either (a) vacant, (b) occupied by a white stone, or (c) occupied by a black stone. Just like a human player, the system starts from the raw board data and successively makes higher-level data structures — stones, strings (a string is connected stones of the same color), groups (strings of the same color that are close to each other), families (loosely connected groups), etc. —, and then determines their attributes (potential value, area of surrounded territory, etc.) in the recognition phase.

- Candidate Move Generation

  The system has *candidate knowledge* which generates the coordinate and evaluation value of a candidate move. To decide the next move, many candidates are listed by executing tasks invoked from candidate knowledge. GOG has 12 kinds of the candidate knowledge (JOSEKI, Edge, DAME, Invasion, Spheres' Contact Point, Capture/Escape, Cut/connect, Enclose/Escape, etc.).

- Next Move Decision

  The local adjustment for candidates rearranges disharmonies between the different candidate knowledges. Then, the system sums the total proposed values of candidates at each point on the board. The system selects the one with the highest value as the next move and plays it.

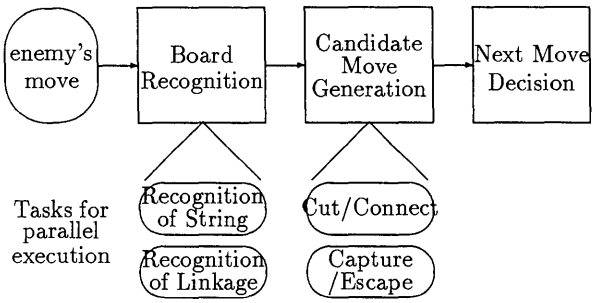


Figure 19: Outline of Process in The Parallel GOG

Parallel Processing In GOG, one of the processors of the Multi-PSI serves as a manager processor, and the rest act as worker processors. The next move decision process is made on the manager processor, which also distributes tasks to worker processors.

When the system receives the enemy's move, it recognizes the board configuration and generates candidate moves. In those processes, it picks up large tasks such as local searches, which check whether a string to be captured or not, and dispatches the worker processors. The results are sent to the manager processor which, then, decides the next move based on those results.

Flying Corps To improve the strength of the system considerably while retaining its real-time response, we proposed the concept of *flying corps.*

This idea is to find the tasks which are important but don't have to be solved before the next move and to make flying corps processes execute these tasks. The system which incorporates the flying corps idea consists of main corps processes and flying corps processes (Figure 20). A flying corps process and a main corps process are assigned to the same processor. Main corps processes consist of a manager and workers and flying corps processes use the same manager and workers. Main corps processes execute necessary tasks to operate by go rules and tasks to maintain their strength.

Main corps processes have a higher priority than flying corps processes. Flying corps processes notify task completion to a flying corps manager process when the dispatched task is completed (which may be several moves after the initiation of the task). Whenever the main corps tasks are finished, the manager process of main corps will collect the results of finished tasks on flying corps processes. With those results and the results by main corps worker processes, the system decides on the next move. The time to decide the next move depends only on the main corps processes.

Flying corps processes execute these tasks independently from the immediate next move decision process (in main corps processes). When the opponent is thinking of the next move, the flying corps processes keep on running. When a local situation, which caused tasks for flying corps, will be changed by some later move, these tasks are aborted.

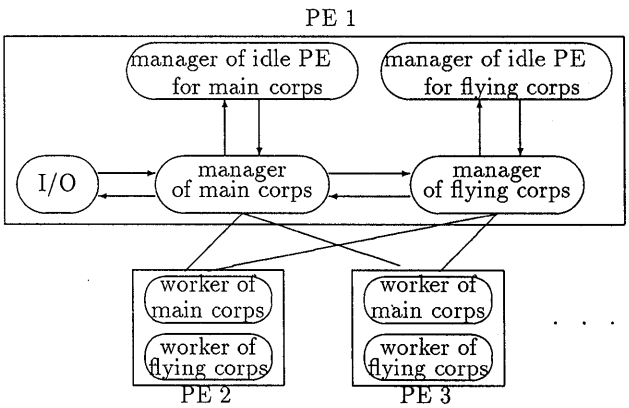


Figure 20: Configuration of System

3.7.3 Result

Table 1 shows the GOG's performance in parallel execution. From these results, the parallel execution shortens the processing time in go. The strength of GOG, including the flying corps idea, is now under evaluation.

We have been developing sequential GOG. The object is to test the new algorithm ideas of recognition, candidate knowledge, and next move decision. Last November, the sequential GOG and seven other computer go programs including last year's top five programs, participated in the tournament at the Game Playing System Workshop. The result of our sequential GOG was 2 wins and 3 loses. It shows that GOG is a top-class computer go-program. In human terms, the current system is stronger than an entry level human go player, but considerably weaker than an average player.

Table 1: Speedup in Parallel Execution

1st of final match, 13th Kisei tournament			
Stage	1 PE	4 PE	16 PE
30th move	1.0	3.3	5.1
90th move	1.0	3.4	5.3
180th move	1.0	3.7	7.5
5th of final match, 13th Meijin tournament			
Stage	1 PE	4 PE	16 PE
30th move	1.0	3.2	5.4
90th move	1.0	3.4	5.6
180th move	1.0	3.6	5.9

4 Overview of Application Programs (2)

4.1 Co-HLEX: Co-operative Recursive LSI Layout Problem Solver

LSI layout is one of the greatest problems requiring massive computation power. Also, the development and enhancement of a layout system consumes huge amounts of programmers labor. In the development of Co-HLEX, the development of a parallel algorithm as well as the possibility of more elegant program descriptions were investigated. The classical divide and conquer algorithm works well while subproblems correlate weakly. For LSI layout, this is not so. Neighboring modules should have abutting shapes and wires to avoid dead spaces. The concurrent co-operation mechanism among processes offered by FGCS paradigm might be an effective means to solve this problem.

An overview of Co-HLEX is given in Figure 21.

The problem-solving kernel is a quadtree-shaped process network called CMPN that generates a chip layout. Before layout generation, each node of CMPN contains circuit data including the module name, the module property, a list of net names connecting this module to others, and a list of sub-circuit names. After the layout is generated, layout data are added to each node: the template name (layoutframe) used to slice the node, the enveloping rectangle size, the list of adopted wiring pattern names for each net, etc.

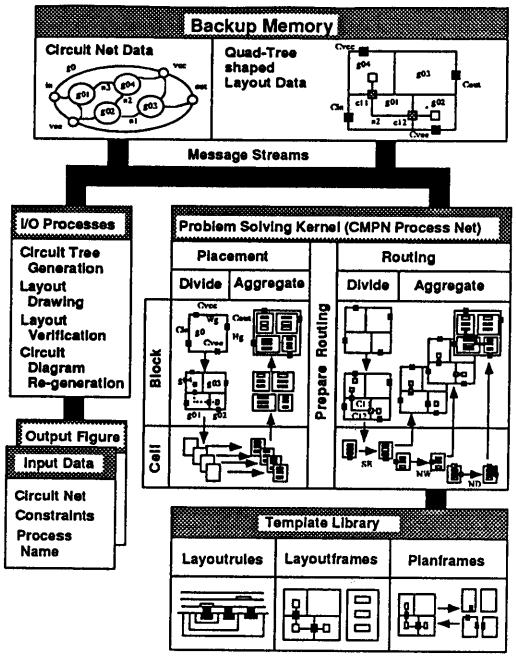


Figure 21: Overview of Co-HLEX

A recursive algorithm called HRCTL (Hierarchical Recursive Concurrent Theorem) was developed. This algorithm performs the layout by the following steps.

Placement A placement message containing a list of planned shape and planned peripheral connector placements is sent to the top node of CMPN from the co-ordination process. Then a set of recursive placement actions is performed by CMPN processes. In top-down processing, each non-terminal node is sliced by using an appropriate layoutframe picked up from the template library. Reaching the leaf node, an appropriate layoutframe defining the cell geometry is chosen. In bottom-up processing, the layouts of lower level children are aggregated to form a parent layout.

Wiring Non-terminal power supply nets Vcc and Vee are wired first, because they interfere with the wiring of signal nets. Non-terminal signal nets are then wired. Then, a set of recursive wiring actions is performed by CMPN. For each net of the non-terminal node, the existence range (CERW) of all the peripheral connectors of the net are first reduced, then an appropriate wiring pattern is selected from the wiring pattern list attached to the layoutframe chosen before. At each point where the chosen pattern crosses the sub-slice border line, an induced connector is introduced. This is used as a peripheral connector by the adjacent sub-slices in the subsequent recursion. Recursion terminates at each leaf node, with each CERW reduced to the magnitude of cell height or width. Lastly, the nets in cells are wired (SE-wiring, NW-wiring, and ND-wiring).

Layout experiments are conducted for bipolar-analog circuits with approximately 1000 modules. The resulting layout realized a compact module placement and wires free of useless bends. By runtime wire abutment cooperation, channel areas used by inter-module patch wires were avoided. This was useful for chip area reduction.

Co-HLEX has a time complexity of roughly $O(N)$, where N is the number of modules in the circuit, as contrasted to a time complexity of nearly $O(N^2)$ for traditional layout systems.

The Co-HLEX program has 1,000 lines in KL1, while traditional implementations typically have more than 100,000 lines of code. The recursive HRCTL algorithm and the modularized streamed-parallel computation model of KL1 both contributed to the size reduction.

4.2 Cooperative Logic Design Expert System on a Multiprocessor

One of the pressing problems of CAD systems is the lack of a means to iterate the cycle of evaluation and redesign until the design satisfies all constraints. Without it, it would be impossible to design a quality circuit with the desired characteristics (area and speed) by looking at the design from a global point of view.

co-LODEX is a cooperative logic design expert system on a multiprocessor, based on an evaluation-redesign mechanism using assumption-based reasoning [Maruyama 1988][Maruyama 1990]. In it, design alternatives are considered as assumptions and constraint violations are viewed as contradictions. Redesign is implemented as contradiction resolution. Justifications for constraint violations, nogood justifications (NJs), play a central role in the mechanism. co-LODEX divides the whole circuit to be designed into subcircuits in advance and designs each subcircuit on each processor to exploit parallel processing. Global evaluation-redesign takes place by processors exchanging design results or NJs. NJs received from other agents help narrow down the search space for an agent in the sense that new NJs made from received NJs enable the agent to prune the search space [Maruyama 1991]. That is the reason why we claim that co-LODEX is cooperative.

co-LODEX inputs a behavioral specification written in a hardware description language, a block diagram of the datapath, and constraints on area and speed. Constraints on area are expressed as inequalities in the gate count, and constraints on speed are expressed as inequalities in the propagation delay. co-LODEX outputs a CMOS standard call netlist that satisfies the constraints.

The resulting netlist can be input to an automatic place-and-route system for CMOS standard cells.

co-LODEX divides the whole circuit to be designed into subcircuits. Each subcircuit is designed by a design agent. Figure 22 shows the five subcircuits for a circuit that solves a second-order differential equation (DiffEQ) and the agents in charge.

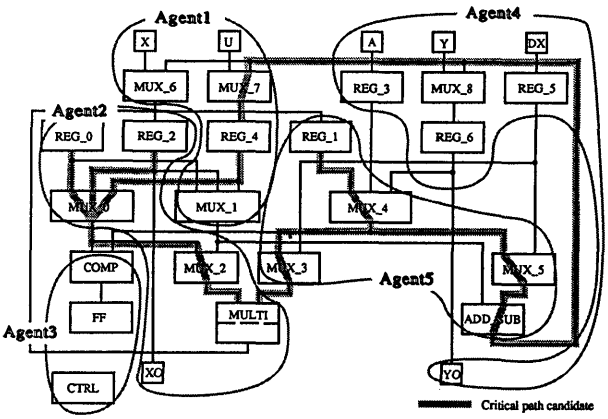


Figure 22: Sub-circuits and agents

Each design agent designs given functional blocks hierarchically using the top-down method. This method keeps splitting functional blocks and subblocks into sub-subblocks until all given blocks are implemented with CMOS standard cells.

Then it counts the number of gates and estimates delays to evaluate the implemented circuit against constraints on area and time. A design agent usually designs its subcircuit independently and in parallel with the other design agents. However, since the design results of the other agents are necessary for evaluation against global constraints, design agents exchange their results every time they design or redesign. A design agent redesigns when it detects a constraint violation for which it is responsible.

co-LODEX was implemented on Multi-PSI in KL1 [Minoda 1992]. Experimental results show that co-LODEX can efficiently carry out global optimization. Design agents correspond to processors on a one-to-one basis. We had one extra processor for distributing the functional blocks to other processors and making statistics. The relation between the number of design agents (1 to 15) and the speedup for a circuit with high uniformity is shown in Figure 23.

4.3 Case-based circuit design system

Recently, much attention has been paid to case-based reasoning (CBR) as a software technology for aquiring large amounts of knowledge easily and utilizing it efficiently. We have researched into a flexible and fast CBR mechanism through upper-level digital circuit design problems.

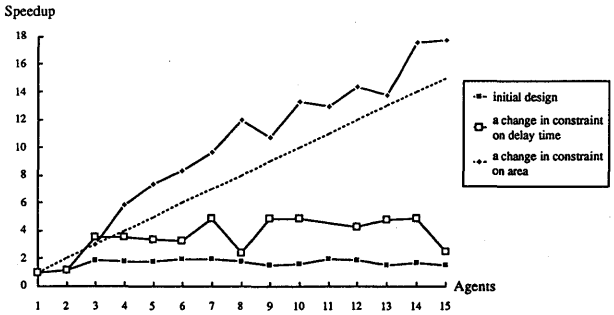


Figure 23: Relation between the number of agents and speedup

We suppose that novice designers, who have knowledge about primitive circuits but lack experiences in design, will use this system to solve application problems that are a little beyond the basic level. This system constructs block diagrams satisfying given specifications by retrieving similar precedent circuits, then modifying and combining them, based only on design cases and knowledge on primitive circuits.

This system features retrieving circuits whose functional structures are similar to the problem's and use a Structure Mapping Engine (SME) [Falkenhainer 86] as a case-retriever.

SME can extract cases structurally similar to the given problem, if higher order relations in given structures are the same between the case and the problem, even if the lower relations and entities are not same. In this system, SME evaluates the similarity of functional hierarchy trees. It, also, evaluates the descriptions of the circuit block functions that represent the hierarchical relations between the primary function and secondary functions that are necessary to realize the primary function. Then, it retrieves the circuits which have the most similar functions as a whole even though the details may be different. For example, when designing a digital clock, SME retrieves a similar circuit which counts the amount of money from the case base, even if there is no digital clock circuit.

Figure 24 shows the configuration of this system. We describe the design process briefly below.

Firstly, *Analyzer* analyzes the input specs to create functional hierarchy trees along the data flow and detailed specs for the given problem. Secondly, *Retriever* retrieves the cases which have similar functional hierarchy trees to the problems with SME. Thirdly, *Adaptor* checks whether the detail specs are the same between

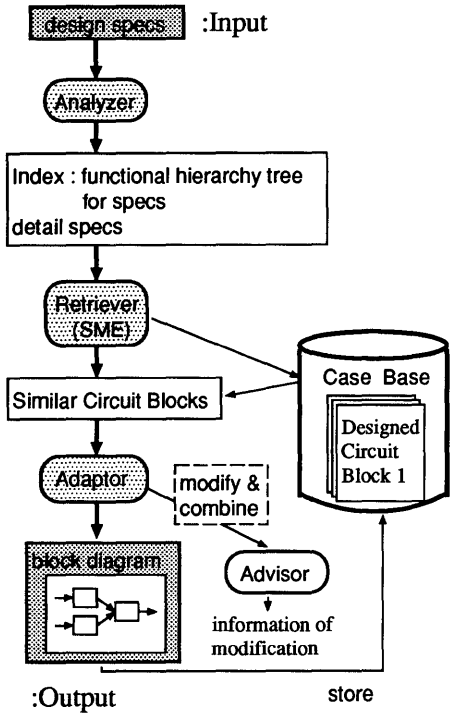


Figure 24: Configuration of case-based circuit design system

the retrieved case and the problem. When different, adaptor checks the possibility of modifying detail specs, then combines the retrieved cases which have confirmed adaptability to the given problem. In this phase, SME also predicts design failures and recovers from them, and those failure recoveries are reported via Advisor. Finally, the system outputs block diagrams corresponding to the combined cases. Users evaluate the output block diagram and, if it is suitable, the problem and the solution are stored in the case base as a new case.

We confirmed that non-stereotyped circuits are actually designed with this approach in mind; i.e. a digital clock with the additional function of sensing temperature can be an air-conditioner performance monitor.

Through experiments we also confirmed the effectiveness of the CBR method with SME. SME, however, has very high running costs because of its structural matching process which includes the combination problem. For this problem, we made SME programs parallel with the multi-level load balancer, and, with the 64 PE of Multi PSI, we obtained 10-fold speedup.

4.4 High Level Synthesis by Parallel Rule-based Annealing

Figure 25 describes the process flow of High Level Synthesis (HLS). LSI behavior descriptions written in a Pascal-like language (Paspec) are parsed and converted to a schedule table. The schedule table describes when each expression is executed and by which ALU. It corresponds to a datapath circuit. The problem finding the lowest cost configuration in the schedule table. The cost is the sum of the chip area and the execution speed. It is a typical combinatorial optimization problem (COP).

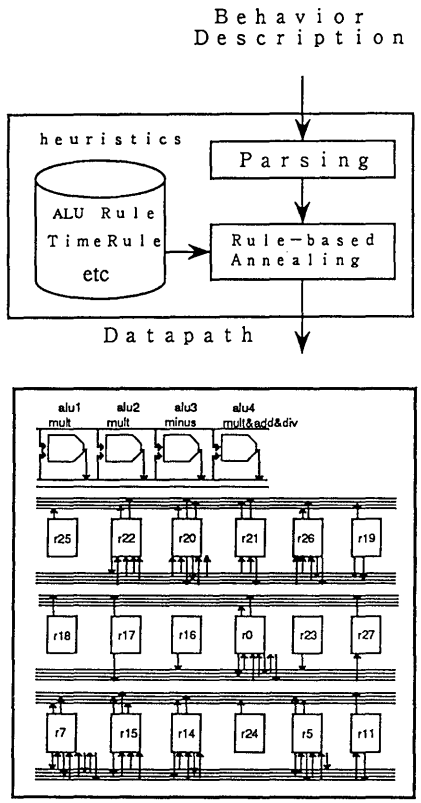


Figure 25: the process flow of HLS

Parallel Rule-Based Annealing Simulated annealing (SA) can be used to find a near global minimum in a COP, but it requires a huge number of iterations. Heuristic algorithms are faster, but the solutions are prone to capture in local minima. The *rule-based annealing (RA)* algorithm was developed, which has intermediate characteristics between the two. In each iteration step, the RA algorithm generates candidates of the next schedule table configuration by using not only random conversion but conversions using heuristic rules. The rule is selected probabilistically and the selection probability of the rule alters the temperature changes. The higher the acceptance rate of the candidate is, the higher the selection probability of the rule.

A parallel RA algorithm was then designed. The system consists of one master processor and a number of

slave processors. Each processor runs the rule-based annealing independently at the same temperature, generating different sequences of configurations. At the beginning of annealing at a temperature, the master processor classifies the slave processors into a higher cost group and a lower cost group based on the cost of configuration. The annealing process continues until there is little difference in the cost distributions of the the two groups, at which time the equilibrium state is considered to have been reached. This contributes to the shortening of annealing steps at high temperatures. At low temperatures, configurations judged to be trapped in local minima are abandoned and are replaced by better configurations in other processors.

The parallel RA algorithm was implemented on a Multi-PSI with 16 processors. Figure 26 shows the experimental results. The RA algorithm was 4 times faster than the SA algorithm. The parallel RA was 8 times faster than the sequential RA. The effectiveness of the parallel RA algorithm was thus experimentally proven.

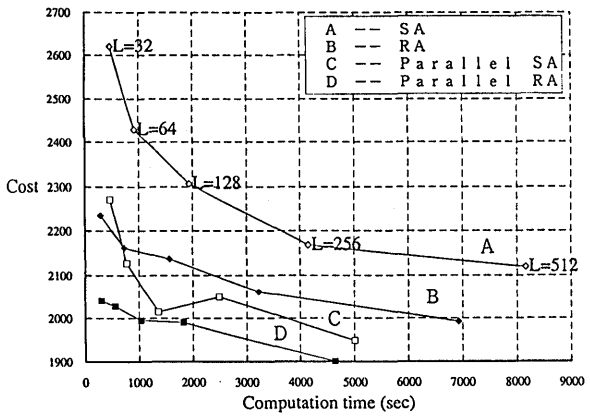


Figure 26: Experimental Result

4.5 Design Supporting System based on Deep Reasoning

In design, there are many cases in which a designer does not directly design a new device, but rather, changes or improves an old device. Sometimes a designer only changes the parameters of components in a device to satisfy the requirements. The designer, in such cases, knows the structure of the device, and needs to determine the new values of the components. This is common in electronic circuits. Desq (Design supporting system based on qualitative reasoning) determines valid ranges of the design decisions using qualitative reasoning.

Desq uses an envisioning mechanism, which, by using qualitative reasoning, determines all possible behaviors of a system. However, the qualitative reasoning of Desq is different from ordinary qualitative reasoning, because it can deal with quantities both qualitatively and quantitatively. Accordingly, Desq may be able to determine quantitative ranges, if parameters are given as quantitative values.

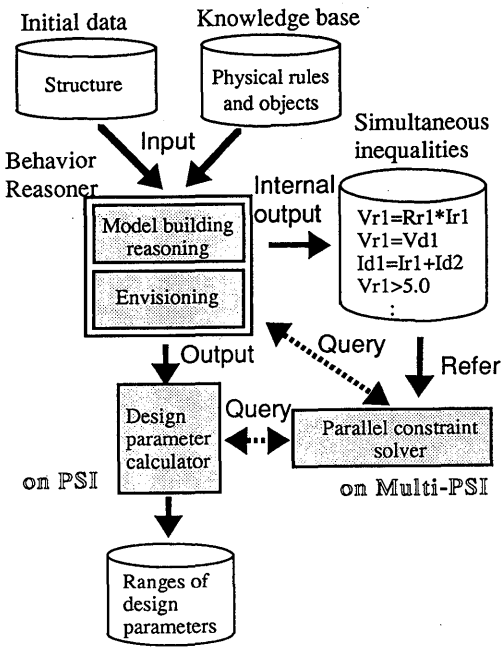


Figure 27: System organization

The system organization of Desq is shown in Figure 27. Desq consists of three subsystems:

Behavior reasoner

This subsystem is based on a qualitative reasoning system. Its model building reasoning part builds simultaneous inequalities from initial data using definitions of physical rules and objects. The simultaneous inequalities are a model of a target system. The envisioning part derives all possible behaviors.

Design parameter calculator

This subsystem calculates ranges of design parameters undefined in initial data.

Parallel constraint solver

This subsystem solves simultaneous inequalities. It is written in KL1 and is executed on a parallel inference machine.

Desq finds the valid ranges of design parameters as follows:

(1) Perform envisioning with design parameters whose values are undefined in initial data,

(2) Select preferable behaviors from possible behaviors found by envisioning,

(3) Calculate the ranges of the design parameters that give preferable behaviors.

As an experiment, Desq successfully determined the valid range of resistance Rb in the DTL circuit in Figure 28.

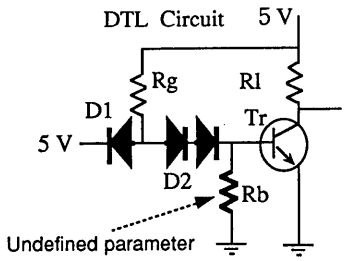


Figure 28: DTL circuit

4.6 A Diagnostic and Control Expert System Based on a Plant Model

Currently in the field of diagnosis and control of thermal power plants, the trend in systems is that the more intelligent and flexible they become, the more knowledge they need. As for knowledge, conventional diagnostic and control expert systems are based on heuristics stored a priori in knowledge bases. So, they cannot deal with unforeseen events when they occur in a plant. Unforeseen events are abnormal situations which were not expected when the plant was designed. To overcome this limitation, we have focused on model-based reasoning and developed a diagnostic and control expert system based on a plant model.

The system (Figure 29) consists of two subsystems: the *Shallow Inference Subsystem (SIS)* and the *Deep Inference Subsystem (DIS)*.

The *SIS* is a conventional plant control system based on heuristics, namely shallow knowledge for plant control. It selects and executes plant operations according to the heuristics stored in the knowledge base. The *Plant Monitor* detects occurrences of unforeseen events, and then activates the *DIS*. The *DIS* utilizes various kinds of models to realize the thought processes of a skilled human operator and to generate the knowledge for plant control to deal with unforeseen events. It consists of the following modules: the *Diagnosor*, the *Operation-Generator*, the *Precondition-Generator*, and the *Simulation-Verifier*. The *Diagnosor* utilizes the *Qualitative Causal Model* for plant process parameters to diagnose unforeseen events. The *Operation-Generator* generates the operations that deal with these unforeseen events. It utilizes the *Device Model* and the *Operation Principle Model*. The *Precondition-Generator* generates the preconditions of each operation generated by the *Operation-Generator*, and, as a result, generates rule-based knowledge for plant control. The *Simulation-Verifier* predicts the plant behavior that will be observed when the plant is operated according to the generated knowledge. It utilizes the *Dynamics Model*, verifies the knowledge using predicted plant behavior, and gives feedback to the *Operation-Generator*, if necessary.

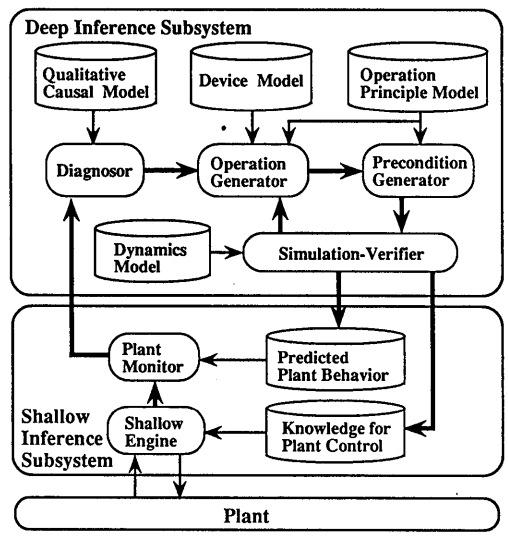


Figure 29: System Overview

The knowledge generated and verified by the *DIS* is transmitted to the *SIS*. The *SIS*, then, executes the plant operations accordingly, and, as a result, the unforeseen events should be taken care of.

We have implemented the system on Multi-PSI. To realize a rich experimental environment, we have also implemented a plant simulator on a mini-computer. Both computers are linked by a data transmission line. We have incorporated both a device and a dynamics model for each device of a thermal power plant (to a total of 78). We summarize the experimental results as follows.

- The *DIS* could generate plan control knowledge to deal with unforeseen events.
- The *SIS* executed plant operators according to the generated knowledge and could deal with unforeseen events.
- We have demonstrated a fivefold improvement in reasoning time by using Multi-PSI with 16 processor elements.

4.7 Adaptive Model-Based Diagnostic System

Though traditional rule-based diagnostic approaches that use symptom-failure association rules have been incorporated by many current diagnostic systems, they lack robustness. This is because they cannot deal with unexpected cases not covered by the rules in its knowledge base. On the other hand, model-based diagnostic systems that use the behavioral specification of a device are more robust than rule-based expert systems. However, in general, many tests are required to reach a conclusive decision because they lack the heuristic knowledge which human experts usually utilize. In order to solve this problem, a model-based diagnostic system has been developed which is adaptable because of its ability to learn from experience [Koseki *et al.* 1990].

This system consists of several modules as shown in Figure 30. The knowledge base consists of *design knowledge* and *experiential knowledge*. The design knowledge represents a correct model of the target device. It consists of a structural description which expresses component interconnections and a behavior description which expresses the behavior of each component. The experiential knowledge is expressed as a failure probability for each component. The *diagnosis module* utilizes those two kinds of knowledge.

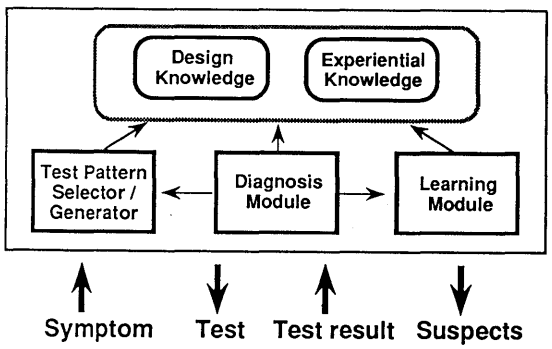


Figure 30: Structure of the System

Figure 31 shows the diagnosis flow of the system. The system keeps a set of suspected components as a suspect-list. It uses an *eliminate-not-suspected* strategy to reduce the number of suspects in the suspect-list, by repeating the test-and-eliminate cycle. It starts by getting an initial symptom. A symptom is represented as a set of target device input signals and an observed incorrect output signal. It calculates an initial suspect-list from the given initial symptoms. It performs model-based reasoning to obtain a suspect-list using a correct design model and an expected correct output signal. To obtain an expected correct output signal for the given inputs, the system carries out simulation using the correct design model.

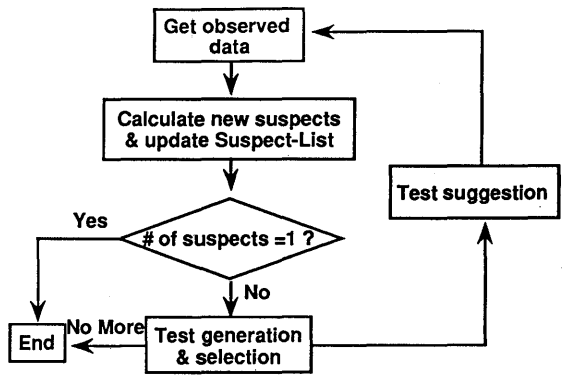


Figure 31: Diagnosis Flow

After obtaining the initial suspect-list, the system repeats a test-and-eliminate cycle, while the number of suspects is greater than one and an effective test exists. A set of tests is generated by the test pattern generator. Among the generated tests, the most cost effective is selected as the next test to be performed. The effectiveness is evaluated by using a minimum entropy technique that utilizes the fault probability distribution. The selected test is suggested and fed into the target device. By feeding the test into the target device, another set of observations are obtained as a test result and are used to eliminate the non-failure components.

Learning Mechanism The performance of the test selection mechanism relies on the preciseness of the presumed probability distribution of components. In order to estimate an appropriate probability distribution from a small amount of observation, the system acquires a presumption tree using minimum description length(MDL) criterion. A description length of a presumption tree is defined as the sum of the code length and the log-likelihood of the model. Using the constructed presumption tree, the probability distribution of future events can be presumed appropriately.

The algorithm is implemented in KL1 language on a parallel inference machine, Multi-PSI. The experimental results show that the 16 PE implementation is about 11 times as fast as the sequential one.

The performance of the adaptive diagnostic system (in terms of the required number of tests) was also examined. The target device was a packet exchange system and its model was comprised of about 70 components. The experimental results show that the number of required tests can be reduced by about 40% on average by using the learned knowledge.

4.8 Motif Extraction System

One of the important issues in genetic information processing is to find common patterns of sequences in the same category which give functional/structural attributes to proteins. The patterns are called *motifs*, in biological terms.

On Multi PSI, we have developed the motif extraction system shown in Figure 32. In this, a motif is represented by stochastic decision predicates and the optimal motif is searched for by the genetic algorithm with the minimum description length(MDL) principle.

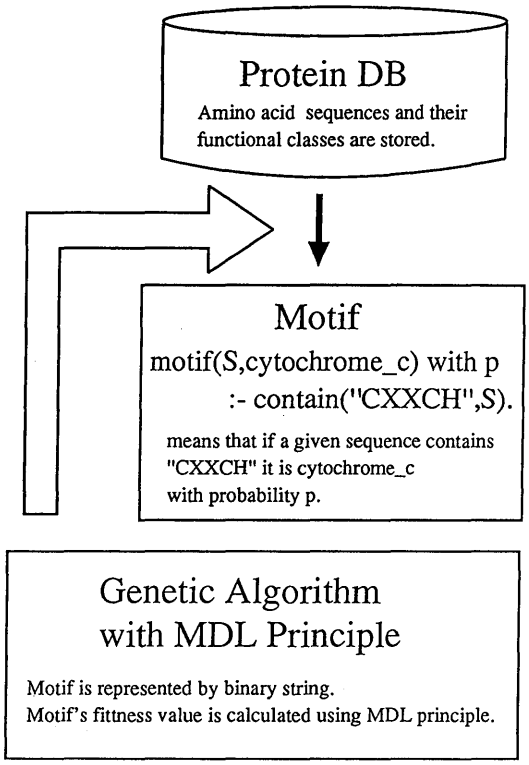


Figure 32: Motif Extraction System

Stochastic Decision Predicate It is difficult to express a motif as an exact symbolic pattern, so we employ the stochastic decision predicate as follows.

```
motif(S,cytochrome_c) with 129/225
  :- contain("CXXCH",S).
motif(S,others) with 8081/8084.
```

This example means that if S contains a subsequence matched to *"CXXCH"*, then S is cytochrome c with probability $\frac{129}{225}$, otherwise S is another protein with probability $\frac{8081}{8084}$.

Minimum Description Length Principle We employ the minimum description length(MDL) principle because it is effective in estimating a good probabilistic model for sample data, including uncertainty avoiding overfitting. The MDL principle suggests that the best stochastic decision predicate minimizes the following value.

> predicate description length + correctness description length

The value of the predicate description length indicates the predicate complexity(i.e. smaller values are better). The value of the correctness description length indicates the likelihood of the predicate(i.e. smaller values are better). Therefore, the MDL principle balances the trade-off between the complexity of motif representation and the likelihood of the predicate to sample data.

Genetic Algorithm The genetic algorithm is a probabilistic search algorithm which simulates the evolution process. We adopt it to search for the optimal stochastic motif, because there is a combinatorially explosive number of stochastic motifs and it takes enormous computation time to find the optimal stochastic motif by exhaustive searches.

The following procedures are performed in order to search for the optimal point of a given function f using the simple genetic algorithm.

1. Give a binary representation that ranges over the domain of the function f
2. Create an initial population which consists of a set of binary strings
3. Update the population repeatedly using selection, crossover, and mutation operators
4. Pick up the best binary string in the population after certain generations

We apply the simple genetic algorithm to search for the optimal motif representation. Each motif is represented by a 120-bit binary string, with each bit corresponding to one pattern (e.g. *"CXXCH"*). The 120-bit binary string represents the predicate whose condition part is the conjunction of the patterns containing the corresponding bits.

Table 2 is the result of applying the motif extraction system to Cytochrome c in the Protein Sequence Database of the National Biomedical Research Foundation. This table shows the extracted motifs and their description lengths. CL is a description length of motif complexity, PL is a description length of probabilities, and DL is a description length of motif correctness.

Table 2: cytochrome c

Motif	Compared	Matched	Correct
CXXCH	8309	225	129
others	8084	8084	8081

Description Length 286.894 (CL = 16.288, PL = 10.397, DL = 260.209)

5 Performance Analysis of Parallel Programs

5.1 Why Performance Analysis?

Along with the development of various application programs, we have been conducting a study of the performance of parallel programs in a more general framework. The main concern is the performance of parallel programs that solve large-scale knowledge information processing problems on large-scale parallel inference machines.

Parallel speedup comes from decomposing the whole problem into a number of subproblems and solving them in parallel. Ideally, a parallelized program would run p times faster on p processors than on one processor. There are, however, various overhead factors, such as load imbalance, communication overhead, and (possible) increases in the amount of computation. Knowledge processing type programs are "non-uniform" in (1) that the number and size of subproblems are rarely predictable, (2) that there can be random communication patterns between the subproblems, and (3) that the amount of total computation can depend on the execution order of subproblems. This makes load balancing, communication control, and scheduling important and nontrivial issues in designing parallel knowledge processing programs.

The overhead factors could make the effective performance obtained by actually running those programs far worse than the "peak performance" of the machine. The performance gap may not be just a constant factor loss (e.g., 30 % loss), but could widen as the number of processors increases. In fact, in poorly designed parallel programs, the effective-to-peak performance ratio can approach zero as the number of processors increases without limit.

If we could understand the behavior of the various overhead factors, we would be able to evaluate parallel programs, identify the most serious bottlenecks, and possibly, remove them. The ultimate goal is to push the horizon of the applicability of large-scale parallel inference machines into a wide variety of areas and problem instances.

5.2 Early Experiences

As the first programs to run on the experimental parallel inference machine Multi-PSI, four programs were developed to solve relatively simple problems. These were demonstrated at the FGCS'88 conference [Ichiyoshi 1989]. They are:

Packing Piece Puzzle (Pentomino)

A rectangular box and a collection of pieces with various shapes are given. The goal is to find all possible ways to pack the pieces into the box. The puzzle is often known as the Pentomino puzzle, when the pieces are all made up of 5 squares. The program does a top-down OR-parallel all solution search.

Shortest Path Problem

Given a graph, where each edge has an associated nonnegative cost, and a start node in the graph, the problem is to find the lowest cost path from the start node to every node in the graph (single-source shortest path problem). The program performs a distributed graph algorithm. We used square grid graphs with randomly generated edge costs.

Natural Language Parser

The problem is to construct all possible parse trees for an English sentence. The program is a PAX parser [Matsumoto 1987], which is essentially a bottom-up chart parsing algorithm. Processes represent chart entries, and are connected by message streams that reflect the data flow in the chart.

Tsumego Solver

A Tsumego problem is to the game of *go* what the checkmate problem is to the game of chess. The black stones surrounding the white stones try to capture the latter by suffocating them, while the white tries to survive. The problem is finding out the result assuming that black and white do their best. The result is (1) white is captured, (2) white survives, or (3) there is a tie. The program does a parallel alpha-beta search.

In the Pentomino program, the parallelism comes from concurrently searching different parts of the search tree. Since disjoint subtrees can be searched totally independently, there is no communication between search subtasks or speculative computation. Thus, load balancing is the key factor in parallel performance. In the first version, we implemented a dynamic load balancing mechanism and attained over 40-fold speedup using 64 processors. The program starts in a processor called the master, which expands the tree and generates search subtasks. Each of the worker processors requests the master processor for a subtask in a demand-driven fashion (i.e.,

it requests a subtask when it becomes idle). Later improvement of data structures and code tuning led to better sequential performance but lower parallel speedup. It was found that the subtask generation throughput of the master processor could not keep up with the subtask solution throughput of the worker processors. A multi-level subtask allocation scheme was introduced, resulting in 50 fold speedup on 64 processors [Furuichi *el al.* 1990].

The load balancing mechanism was separated from the program, and was released to other users as a utility. Several programs have used it. One of them is a parallel iterative deepening A* program for solving the Fifteen puzzle. Although the search tree is very unbalanced because of pruning with a heuristic function, it attained over 100 fold speedup on a 128-processor PIM/m [Wada *et al.* 1992].

The shortest path program has a lot of inter-*process* communication, but the communication is between neighboring vertices. A mapping that respects the locality of the original grid graph can keep the amount of inter-*processor* communication low. A simple mapping, in which the square graph was divided into as many subgraphs as there are processors, maximized locality. But the parallel speedup was poor, because the computation spread like a wavefront, making only some of the processors busy at any time during execution. By dividing the graph into smaller pieces and mapping a number of pieces from different parts of the graph, processor utilization was increased [Wada and Ichiyoshi 1990].

The natural language parser is a communication intensive program with a non-local communication pattern. The first static mapping of processes showed very little speedup. It was rewritten so that processes migrate to where the necessary data reside to reduce inter-processor communication. It almost halved the execution time [Susaki *et al.* 1989].

The Tsumego program did parallel alpha-beta searches up to the leaf nodes of the game tree. Sequential alpha-beta pruning can halve the effective branching factor of the game tree in the best cases. Simply searching different alternative moves in parallel loses much of this pruning effect. In other words, the parallel version might do a lot of redundant speculative computation. In the Tsumego program, the search tasks of candidate moves are given execution priorities according to the estimated value of the moves, so as to reduce the amount of speculative computation [Oki 1989].

Through the development of these programs, a number of techniques were developed for balancing the load, localizing communication, and reducing the amount of speculative computation.

5.3 Scalability Analysis

A deeper understanding of various overheads in parallel execution requires the construction of models and analysis of those models. The results form a robust core of insight into parallel performance.

The focus of the research was the scalability of parallel programs. Good parallel programs for utilizing large-scale parallel inference machines have performance that scales, i.e., the performance increases in accordance with the increase in the number of processors. For example, two-level load balancing is more scalable than single-level load balancing, because it can use more processors. But deciding how scalable a program is requires some analytical method.

As a measure of scalability, we chose the *iso-efficiency function* proposed by Kumar and Rao [Kumar *et al.* 1988]. For a fixed problem instance, the efficiency of a parallel algorithm (the speedup divided by the number of processors) generally decreases as the number of processors increases. The efficiency can often be regained by increasing the problem size. The function $f(p)$ is defined as an isoefficiency function if the problem size (identified with the sequential runtime) has to increase as $f(p)$ to maintain a given constant efficiency E as the number of processors p increases. An isoefficiency function grows at least linearly as p increases (lest the subtask size allocated to each processor approaches zero). Due to various overheads, isoefficiency functions generally have strictly more than linear growth in p. A slow growth rate, such as $p \log p$, in the isoefficiency function would mean a desired efficiency can be obtained by running a problem with a relatively small problem size. On the other hand, a very rapid growth rate such as 2^p would indicate that only a very poor use of a large-scale parallel computer would be possible by running a problem with a realistic size.

On-demand load balancing was chosen first for analysis. Based on a probabilistic model and explicitly stated assumptions on the nature of the problem, the isoefficiency functions of single-level load balancing and multi-level load balancing were obtained. In a deterministic case (all subtasks have the same running time), the isoefficiency function for single-level load balancing is p^2, and that for two-level load balancing is $p^{3/2}$. The dependence of the isoefficiency functions on the variation in subtask sizes was also investigated, and it was found that if the subtask size is distributed according to an exponential distribution, a $\log p$ (respectively, $(\log p)^{3/2}$) factor is added to the isoefficiency function of single-level (respectively, two-level) load balancing. The details are found in [Kimura *et al.* 1991].

More recently, we studied the load balance of distributed hash tables. A distributed hash table is a parallelization of a sequential hash table; the table is divided into subtables of equal size, each one of which is allocated to each processor. A number of search operations for the table can be processed concurrently, resulting in increased throughput. The overhead comes mainly from load imbalance and communication overhead. By allo-

cating an increasing number of buckets (= subtable size) to each processor, the load is expected to be improved. We set out to determine the necessary rate of increase of subtable size to maintain a good load balance. A very simple static load distribution model was defined and analyzed, and the isoefficiency function (with regard to load imbalance) was obtained [Ichiyoshi *et al.* 1992]. It was found that a relatively moderate growth in subtable size q ($q = \omega((\log p)^2)$) is sufficient for the average load to approach perfect balance. This means that the distributed hash table is a data structure that can exploit the computational power of highly parallel computers with problems of a reasonable size.

5.4 Remaining Tasks

We have experimented with a few techniques for making better use of the computational power of large-scale parallel computers. We have also conducted a scalability analysis for particular instances of both dynamic and static load balancing. The analysis of various parallelizing overheads and the determination of their asymptotic characteristics gives insight into the nature of large-scale parallel processing, and guides us in the design of programs which run on large-scale parallel computers.

However, what we have done is a modest exploration of the new world of large-scale parallel computation. The analysis technique must be expanded to include communication overheads and speculative computation. Now that PIM machines with hundreds of processors have become operational, the results of asymptotic analysis can be compared to experimental data and their applicability can be evaluated.

6 Summary of Parallel Application Programs

We have introduced overviews on parallel application programs and results of performance analysis. We will summarize knowledge processing and parallel processing using PIMs/KL1.

(1) Knowledge Processing by PIM/KL1

We have developed parallel intelligent systems such as CAD systems, diagnosis systems, control systems, a game system, and so on. Knowledge technologies used in them are the newest, and these systems are valuable from viewpoint of AI applications, too. Usually, as these technologies need much computation time, it is impossible to solve large problems using sequential machines. Therefore, these systems are appropriate to evaluate effectiveness of parallel inference.

We have already been experienced in knowledge processing by sequential logic programming languages. Therefore, we have got accustomed to developing programs in KL1 in a short time. Generally, to develop parallel programs, programmers have to consider the synchronization of each modules. This is troublesome and often causes bugs. However, as KL1 has automated mechanisms to synchronize inferences, we were able to develop parallel programs in a relatively short period of time as follows.

Program	Size	man*month
Logic Simulator	8 k	3
Placement		
(KL1)	4 k	4
(ESP†)	8 k	4
Routing	4.9 k	2
Alignment by 3-DP	7.5 k	4
Alignment by SA	3.7 k	2
Folding Simulation	13.7 k	5
Legal Reasoning		
(Rule-based engine)	2.5 k	3
(Case-based engine)	2 k	6
Go Playing Game	11 k	10

†: An extended Prolog for system programming.

In those cases where the program didn't show high performance, we had to consider another process model in regards to granularity of parallelism. Therefore, we have to design the problem solution model in more detail than when developing it on sequential machines.

(2) Two types of Process Programming

The programming style of KL1 is different from that of sequential logic programming language. A typical programming style in KL1 is *process programming*. A *process* is an object which has *internal states* and *procedures* to manipulate those internal states. Each process is connected to other processes by *streams*. Communication is through these streams. A process structure can be easily realized in KL1 and many problem solving techniques can be modeled by process structures.

We observed that two types of KL1 process structure are used in application programs.

1. **Static process structure**

   The first type of process structure is a static one. In this, a process structure for problem solving is constructed, then, information is exchanged between processes. The process structure doesn't change until the given problem is solved. Most distributed algorithms have a static process structure. The majority of application programs belong to this type.

   For example, in the Logic Simulator, an electrical circuit is divided into sub circuits and each sub circuit

is represented as a process (Figure 3). In the Protein Sequence Analysis System, two protein sequences are represented as a two dimensional network of KL1 processes (Figure 9). In the Legal Reasoning System, the lefthand side of a case rule is represented as a Rete-like network of KL1 processes (Figure 17). In co-LODEX, design agents are statically mapped onto processors (Figure 22).

2. **Dynamic process structure**

   The second type of process structure is a dynamic one. The process structure changes during computation. Typically, the toplevel process forks into subprocesses, each subprocess forks into subsubprocesses, and so on (Figure 33). Usually, this process structure corresponds to a search tree. Application programs such as Pentomino, Fifteen Puzzle and Tsumego belong to this type.

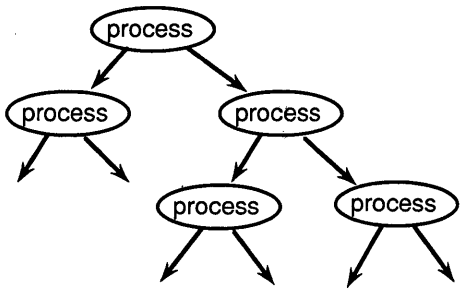


Figure 33: A search tree by a dynamic process structure

(3) New Paradigm for Parallel Algorithms

We developed new programming paradigms while designing parallel programs. Some of the parallel algorithms are not just parallelizations of sequential algorithms, but have desirable properties not present in the base algorithm.

In combinatorial optimization programs, a parallel simulated annealing (SA) algorithm (used in the LSI cell placement program and MASCOT), a parallel rule-based annealing (RA) algorithm (used in the High Level Synthesis System), and a parallel genetic algorithm (GA) (used in the Motif Extraction System) were designed.

The parallel SA algorithm is not just a parallel version of a sequential SA algorithm. By statically assigning temperatures to processors and allowing solutions to move from processor to processor, the solutions compete for lower temperature processors: a better solution has a high possibility of moving to a lower temperature. Thus, the programmer is freed from case-by-case tuning of temperature scheduling. The parallel SA algorithm is also time-homogeneous, an important consequence of which is it does not have the problem in sequential SA that the solution can be irreversibly trapped in a local minimum at a low temperature.

In the parallel RA algorithm, the distribution of the solution costs are monitored, and used to the judge whether or not the equilibrium state has been reached.

In the go-playing program, the flying corps idea suited for real-time problem solving was introduced. The task of the flying corps is to investigate the outcome of moves that could result in a potentially large gain (such as capturing a large opponent group or invasion of a large opponent territory) or loss. The investigation of a possibility may take much longer time than allowed in real-time move making and cannot be done by the *main corps*.

(4) Performance by Parallel Inference

Some application programs exhibited high performance by parallel execution, such as up to 100-fold speedup using 128 processors. Examples include the logic simulator (LS) (Figure 4), the legal reasoning system (LR) (Figure 18), and MGTP which is a theorem prover developed by the fifth research laboratory of ICOT[Fujita *et al.* 1991] [Hasegawa *et al.* 1992]. Understandably, these are the cases where there is a lot of parallelism and parallelization overheads are minimized. The logic simulator (LS), the legal reasoning system (LR), and MGTP have high parallelism coming from the data size (a large number of gates in the logic simulator and a large number of case rules in the legal reasoning system) or problem space size (MGTP). A good load balance was realized by static even data allocation (LS, LR), or by dynamic load allocation (MGTP). Either communication locality was preserved by process clustering (LS), or communication between independent subtasks is small (rule set division in LR or OR-parallel search in MGTP).

(5) Load Distribution Paradigm

In all our application programs, programs with a static process structure used a static load distribution, while programs with a dynamic process structure used semi-static or dynamic load distribution.

In a program with a static process structure, a good load balance can usually be obtained by assigning roughly the same number of processes to each processor. To reduce the communication overhead, it is desirable to respect the locality in the logical process structure. Thus, we first divide the processes into clusters of processes that are close to each other. Then, the clusters are mapped onto the processors. This direct cluster-to-processor mapping may not attain good load balance, since, at a given point in computation, only part of the process structure has a high level of computational activity. In such a case, it is better to divide the process

structure into smaller clusters and map a number of clusters that are far apart from each other on one processor. This multiple mapping scheme is adopted in the shortest path program and the logic simulator. In the three dimensional DP matching program, a succession of alignment problems (sets of three protein sequences to align) are fed into the machine and the alignment is performed in a pipelined fashion, keeping most processors busy all the time.

In a program with a dynamic process structure, newly spawned processes can be allocated to processors with a light computational load to balance the load. To maintain low communication overhead, only a small number of processes are selected as candidates of load distribution. For example, in a tree search program, not all search substasks but only those at certain depths are chosen for interprocessor load allocation. The Pentomino puzzle solver, the Fifteen puzzle solver and the Tsumego solver use this on-demand dynamic load balancing scheme.

(6) Granularity of Parallelism

To obtain high performance by parallel processing, we have to consider the granularity of parallelism. If the size of each subtask is small, it is hard to obtain high performance, because parallelization overheads such as process switching and communication are serious. For example, in the first version of the Logic Simulator, the gates of the electrical circuit were represented as processes communicating with each other via streams. The performance of this version was not high because the task for each process was too small. The second version represented subcircuits as processes (Figure 3), and succeeded in improving the performance.

(7) Programming Environment

The first programs to run on the Multi-PSI were developed before the KL1 implementation on the machine had been built. The user wrote and debugged a program on the sequential PDSS (PIMOS development support system) on a standard hardware. The program was then ported to the the Multi-PSI, with the addition of load distribution pragmas. The only debugging facilities on the Multi-PSI were those developed for debugging the implementation itself, and it was not easy to debug application programs with those facilities. Gradually, the PIMOS operating system [Chikayama 1992] added debugging facilities such as an interactive tracing/spying facility, a static code checker that gives warnings on single-occurrence variables which are often simply misspelled, and a deadlock reporting facility. The deadlock reporting facility identifies perpetually suspended goals and, instead of printing out all of them (possibly very many), it displays only a goal that is most upstream in the data flow. It has been extremely helpful in locating the cause of a perpetual suspension (usually, the culprit is a producer process failing to instantiate the variable on which the reported goal is suspended).

Performance monitoring and gathering facility was later added (and is still being enhanced) [Aikawa 1992]. Post-mortem display of processor utilization along the time axis often clearly reveals that one processor is being a bottleneck at a particular phase of computation. The breakdown of processor time (into computing/communicating/idling) can give a hint on how the process structure might be changed to remove the bottleneck.

Sometimes knowledge of KL1 implementation is necessary to interpret the information provided by the facility to tune (sequential as well as parallel) performance. A similar situation exists in performance tuning of application programs on any computers, but the problem seems to be more serious in a parallel symbolic language like KL1. How to bridge the gap between the programmer's idea of KL1 and the underlying implementation remains a problem in performance debugging/tuning.

7 Conclusion

We introduced overviews of parallel application programs and research on performance analysis.

Application programs presented here contain interesting technologies from viewpoint of not only parallel processing but knowledge processing.

By developing various knowledge processing technologies in KL1 and measuring their performance, we showed that KL1 is a suitable language to realize parallel knowledge processing technologies and that they are executed quickly on PIM. Therefore, PIM and KL1 are appropriate tools to develop large scale intelligent systems.

Moreover, we have developed many parallel programming techniques to obtain high performance. We were able to observe their effects actually on the parallel inference machine. These experiences are summarized as guidelines for developing larger application systems.

In addition to developing application programs, the performance analysis group analyzed behaviors of parallel programs in a general framework. The results of performance analysis gave us useful information for selecting parallel programming techniques and for predicting their performance when the problem sizes are scaled up.

The parallel inference performances presented in this paper were measured on Multi-PSI or PIM/m. We need to compare and analyze the performances on different PIMs as future works. We would also like to develop more utility programs which will help us to develop parallel programs, such as a dynamic load balancer other than the multi-level load balancer.

Acknowledgement

The research and development of parallel application programs has been carried out by researchers of the seventh research laboratory and cooperating manufacturers with suggestions by members of the PIC, GIP, ADS and KAR working groups. We would like to acknowledge them and their efforts. We also thank Kazuhiro Fuchi, the director of ICOT, and Shunichi Uchida, the manager of the research department.

References

[Aikawa 1992] S. Aikawa, K. Mayumi, H. Kubo, F. Matsuzawa. ParaGraph: A Graphical Tuning Tool for Multiprocessor Ssytems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992.

[Barton 1990] J. G. Barton, Protein Multiple Alignment and Flexible Pattern Matching. In *Methods in Enzymology, Vol.183* (1990), Academic Press, pp. 626-645.

[Chikayama 1992] Takashi Chikayama. KL1 and PIMOS. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992.

[Date *et al.* 1992] H. Date, Y. Matsumoto, M. Hoshi, H. Kato, K. Kimura and K. Taki. LSI-CAD Programs on Parallel Inference Machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992.

[de Kleer 1986] J. de Kleer. An Assumption-Based Truth Maintenance System, Artificial Intelligence 28, (1986), pp.127-162.

[Doyle 1979] J. Doyle. A Truth Maintenance System. Artificial Intelligence 24 (1986).

[Falkenhainer 86] B. Falkenhainer, K. D. Forbus, D. Gentner. The Structure-Mapping Engine. In *Proc. Fifth National Conference on Artifical Intelligence*, 1986.

[Fujita *et al.* 1991] H. Fujita, et. al. A Model Generation Therem Prover in KL1 Using a Ramified-Stack Algorithm. ICOT TR-606 1991.

[Fukui 1989] S. Fukui. Improvement of the Virtual Time Algorithm. *Transactions of Information Processing Society of Japan*, Vol.30, No.12 (1989), pp. 1547-1554. (in Japanese)

[Furuichi *el al.* 1990] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the Multi-PSI. In *Proc. of PPoPP'90*, 1990, pp. 50–59.

[Goto *et al.* 1988] Atsuhiro Goto *et al.* Overview of the Parallel Inference Machine Architecture. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988.

[Hasegawa *et al.* 1992] Hasegawa, R. *et al.* MGTP: A Parallel Theorem Prover Based on Lazy Model Generation. To appear in *Proc. CADE' (System Abstract)*, 1992.

[Hirosawa *et al.* 1991]
Hirosawa, M., Hoshida, M., Ishikawa, M. and T. Toya, T. Multiple Alignment System for Protein Sequences employing 3-dimensional Dynamic Programming. In *Proc. Genome Informatics Workshop II*, 1991 (in Japanese).

[Hirosawa *et al.* 1992] Hirosawa, H., Feldmann, R.J., Rawn, D., Ishikawa, M., Hoshida, M. and Micheals, G. Folding simulation using Temperature parallel Simulated Annealing. In *Proc. Int. Conf. on Fifth Generation Computer System 1992*, ICOT, Tokyo, 1992.

[Ichiyoshi 1989] N. Ichiyoshi. Parallel logic programming on the Multi-PSI. ICOT TR-487, 1989. (Presented at the Italian-Swedish-Japanese Workshop '90).

[Ichiyoshi *et al.* 1992] N. Ichiyoshi and K. Kimura. Asymptotic load balance of distributed hash tables. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, 1992.

[Ishikawa *et al.* 1991] Ishikawa,M., Hoshida,M., Hirosawa,M., Toya,T., Onizuka,K. and Nitta,K. (1991a) Protein Sequence Analysis by Parallel Inference Machine. *Information Processing Society of Japan, TR-FI-23-2*, (in Japanese).

[Jefferson 1985] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Vol.7, No.3 (1985), pp. 404-425.

[Kimura *et al.* 1991] K. Kimura and K. Taki. Time-homogeneous Parallel Annealing Algorithm. In *Proc. IMACS'91*, 1991. pp. 827-828.

[Kimura *et al.* 1991] K. Kimura and N. Ichiyoshi. Probabilistic analysis of the optimal efficiency of the multi-level dynamic load balancing scheme. In *Proc. Sixth Distributed Memory Computing Conference*, 1991, pp. 145–152.

[Kitazawa 1985] H. Kitazawa. A Line Search Algorithm with High Wireability For Custom VLSI Design, In *Proc. ISCAS'85*, 1985. pp.1035–1038.

[Koseki *et al.* 1990] Koseki, Y., Nakakuki, Y., and Tanaka, M., An adaptive model-Based diagnostic system, In *Proc. PRICAI'90,* Vol. 1 (1990), pp. 104-109.

[Kumar *et al.* 1988] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state space graphs: A summary of results. In *Proc. AAAI-88*, 1988, pp. 122–127.

[Maruyama 1988] F. Maruyama et al. co-LODEX: a cooperative expert system for logic design. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.1299-1306.

[Maruyama 1990] F. Maruyama et al. Logic Design System with Evaluation-Redesign Mechanism. Electronics and Communications in Japan, Part III: Fundamental Electronic Science, Vol. 73, No.5, Scripta Technica, Inc. (1990).

[Maruyama 1991] F. Maruyama et al. Solving Combinatorial Constraint Satisfaction and Optimization Problems Using Sufficient Conditions for Constraint Violation. In Proc. the Fourth Int. Symposium on Artificial Intelligence, 1991.

[Matsumoto 1987] Y. Matsumoto. A parallel parsing system for natural language analysis. In *Proc. Third International Conference on Logic Programming*, Lecture1 Notes on Computer Science 225, Springer-Verlag, 1987, pp. 396–409.

[Matsumoto *et al.* 1992] Y. Matsumoto and K. Taki. Parallel logic Simulator based on Time Warp and its Evaluation. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992.

[Minoda 1992] Y. Minoda et al. A Cooperative Logic Design Expert System on a Multiprocessor. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*,ICOT,Tokyo, 1992.

[Nakakuki *et al.* 1990] Nakakuki, Y., Koseki, Y., and Tanaka, M., Inductive learning in probabilistic domain, In *Proc. AAAI-90,* Vol. 2 (1990), pp. 809-814.

[Needleman *et al.* 1970] Needleman,S.B. and Wunsch,C.D. A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. *J. of Mol. Biol.*, **48** (1970), pp. 443-453.

[Nitta *et al.* 1992] K. Nitta et. al. HELIC-II: A Legal Reasoning System on the Parallel Inference Machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992.

[Oki 1989] H. Oki, K. Taki, S. Sei, and M. Furuichi. Implementation and evaluation of parallel Tsumego program on the Multi-PSI. In *Proc. the Joint Parallel Processing Symposium (JSPP'89)*, 1989, pp. 351–357. (In Japanese).

[Skolnick and Kolinsky 1991] Skolnick, J. and Kolinski,A., Dynamic Monte Carlo Simulation of a New Lattice Model of Globular Protein Folding, Structure and Dynamics, *Journal of Molecular Biology, Vol.221, No2*, pp.499-531.

[Susaki *et al.* 1989] K. Susaki, H. Sato, R. Sugimura, K. Akasaka, K. Taki, S. Yamazaki, and N. Hirota. Implementation and evaluation of parallel syntax analyzer PAX on the Multi-PSI. In *Proc. Joint Parallel Processing Symposium (JSPP'89)*, 1989, pp. 342–350. (In Japanese).

[Uchida *et al.* 1988] Shunichi Uchida et al. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Ueda *et al.* 1978] Ueda, Y., Taketomi, H. and Go, N. (1978) Studies on protein folding, unfolding and fluctuations by computer simulation. A three dimensional lattice model of lysozyme. *Bilpolymers Vol.17* pp.1531-1548.

[Wada and Ichiyoshi 1990] K. Wada and N. Ichiyoshi. A study of mapping of locally message exchanging algorithms on a loosely-coupled multiprocessor. ICOT TR-587, 1990.

[Wada *et al.* 1992]
M. Wada, K. Rokusawa, and N. Ichiyoshi. Parallelization of iterative deepening A* algorithm and its implementation and performance measurement on PIM/m. To appear in Joint Symposium on Parallel Processing JSPP'92 (in Japanese).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Algorithmic & Knowledge Based Methods Do they "Unify" ?

With some Programme Remarks for *UNU/IIST**

Dines Bjørner and Jørgen Fischer Nilsson†

April 1992

Abstract

We examine two approaches to software application development. One is based on the conventional stepwise algorithmic approach typified by the imperative programming language (eg. PASCAL) tradition — but extends it with mathematical techniques for requirements development. The other is the knowledge based systems approach typified by the logic programming (eg. PROLOG) tradition. We contrast techniques and we attempt to find unifying issues and techniques. We propose a *Most "Grand" Unifier* — in the form of a *Partial Evaluator* (ie. *Meta-interpreter* — which establishes relations between the two approaches.

The paper finally informs of the *UNU/IIST*, the United Nations University's International Institute for Software Technology. *UNU/IIST* shall serve especially the developing world. We outline consequences of the present analysis for the work of the *UNU/IIST*.

The Fifth Generation Computer Project

When the first author was invited, in late February, to read this paper at the plenum session of the International Conference on *Fifth Generation Computer Systems* it was expected that ... *UNU/IIST strategies for prompting research and development in the field of computer science, including issues of education, creativity and international collaboration ... and future prospects of computer science ...* would be covered by this presentation.

In accepting the kind invitation the following acknowledgement was expressed: *the decision by MITI to start, 10 years ago, the 5th Generation Computer Project has had enormous and very positive influence on world-wide research, engineering and application of knowledge based computing systems. Japan has thus, as an example, made the world of computing dramatically more professional and exciting.*

The Japanese Fifth Generation Project has focused specifically on what we here list as the second approach.

Structure of Paper

The paper is organised as follows. Section 1 presents an overview of main characteristics of the two approaches. Section 2 informally discusses algorithmic compiler and interpreted versus knowledge based iplementations. Section 3 presents facets of the algorithmic approach, emphasizing mathematical requirements development as a phase prior to algorithmic software development. The section brings a first comparison of the two approaches surveyed in this paper. Section 4 compares Table-by-Table the algorithmic and the knowledge based approaches. Section 5 then attempts to bring ideas on how the approaches relate, ie. can be "unified" at some meta-level.

Section 6 surveys *UNU/IIST*.

Appendix A complements section 3's treatment of requirements development. We have not, in our comparisons, in section 4, taken the additional, algorithmic requirements modelling facets covered in the appendix into account — this is left to future analysis.

1 Algorithms and Knowledge Based Systems

The "classical" approach to software development is based on FORTRAN, ALGOL 60, PASCAL, and more recently also object oriented approaches such as C++. In this style of programming one transforms some abstraction of the problem domain, in several stages of develop-

*Invited paper for the Plenum Session of the International Conference on *Fifth Generation Computer Systems, FGCS'92* ICOT, Tokyo, Japan, June 1–5, 1992.

†Professor Dines Bjørner is Director of *UNU/IIST*: *United Nations University's International Institute for Software Technology*, Apartado (Post office box) 517, Macau, e-mail: `unuiist%uealab%umacmr@hkucnt.hku.hk`. Jørgen Fischer Nilsson is Professor of Knowledge Based Systems at the Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, e-mail: `jfn@id.dth.dk` — from where Dines Bjørner is on an extended leave of absence.

ment, into compiled, execution time and storage space efficient machine code.

References [3, 7, 5] represent the algorithmic approach.

The Knowledge Based Systems (KBS) approach — here also referred to as the Knowledge Engineering (KE) approach — is usually based on the LISP, and more recently on the logic programming (PROLOG) tradition of programming. Here a programming paradigm is followed in which one basically attempts to represent knowledge about the problem domain in such a way that it is this representation which forms the basis for execution time computation.

References [9, 6, 8] are typical of the knowledge based approach.

We here call the "classical" programming paradigm for 'algorithmic' (and by $\mathcal{A}$ we mean Algorithmic Software Development), whilst the other style is referred to as 'knowledge based' (and by $\mathcal{K}$ we mean Knowledge Engineering).

Section 4 contrasts the two styles of programming & programs.

2 Complementing Approaches

General: In creating a database, in the $\mathcal{A}$ approach, one oftentimes imperceptibly transform the knowledge gained, when analysing the application, into some efficient physical storage representation — thereby "compiling" away those knowledge facets that might be useful in heretofore unforeseen contexts.

In creating a knowledge base such generally adaptable knowledge is preserved. Execution time is longer, but the time it takes for humans to adapt to changing circumstances and get it "running" is generally very short.

In either case (of information base creation) the requirements developer spends a long time analysing the domain of interest. In the former case the knowledge gained is specialised wrt. the specific, usually narrow, foreseen application — and hence most is lost wrt. to unforeseen, future, related applications.

Take an example: One can write a software system for the regulation (monitoring & control) of a railway system domain, in either of three ways.

1: The Compiled Algorithmic Approach: $\mathcal{C}$

In the first, using the $\mathcal{A}$ approach, knowledge about the specifics of railways is hidden in the code of the programs: in their compiled constants, and in the structure of the code itself: in the specific composition of statements.

2: The Interpreted Algorithmic Approach: $\mathcal{I}$

Alternatively one can choose to represent the railway system component as a database: all the tracks, the trains, the schedule, etc., are data structures, and computation proceeds through interpretation. The railway system specific interpreter ($\mathcal{I}$) is, however, general in that it is the same one interpreter, written once, that can be used to handle a wide variety of railway systems. Execution is slower than in the former case, but one can more easily introduce changes to the schedule, tracks, trains, etc.

— Compiled and Interpreted Algorithmics: $\mathcal{A}$

The compiled, first version represents a so-called partially evaluated version of the latter case. In either case, the compiled code respectively the interpreter does not reason about train regulation (based on logic), but, typically arithmetically computes based on classical mathematical laws of kinematics and operations research. Thus these laws are "baked" into the compiled code, respectively into the specific interpreter.

3: The Knowledge Based Approach: $\mathcal{K}$

Instead of either of these two approaches one could represent these laws in logical form — in addition to, but in the same form as the representation of the railway system components. "Computation" now proceeds by means of a general meta-interpreter, known as an inference machine ($\mathcal{M}$), the same for a broad range of such systems, not only railway systems, but virtually "anything"!

Review: we shall review this example in section 5 — where we shall use the calligraphic symbols: $\mathcal{C}, \mathcal{I}, \mathcal{A}, \mathcal{K}$.

3 Algorithmic Software Development

The present section very briefly outlines development techniques of the so-called 'algorithmic' approach, and gives a more technical presentation of this approach. [3] specifically addresses the issue of the algorithmic development of embedded, real-time systems.

Algorithmic development consists of two major phases: requirements development and software development.

3.1 Requirements Development

Requirements development, in the algorithmic approach, to us consists of three steps: the informal expression (*narration*) of the problem, requirements modelling and requirements capture. We will only illustrate a single facet, out of several, of the requirmments modelling step. Appendix A surveys other facets.

3.1.1 Narrative

The narrative serves the rôle of establishing the terminology necessary to formulate expectations.

Example: *The railway system has as base, atomic components and component attributes namely those of week days, hours, minutes, station identities, train identities, velocity & acceleration, and track segments. The total railway system component (the "grand state") is then seen as composed from some composition of (some) time components (colloquially referred to, somewhat ambiguously as 'time'), a schedule, tracks and trains. The schedule records train departure and arrival times between listed stations. The tracks are modelled as a graph whose nodes are stations and whose arcs are sequences of track segments of distinct lengths. Trains have a position at a station or at some track segment between two stations, and trains have some kinematics: velocity and acceleration.* □

3.1.2 Requirements Modelling

In requirements modelling we investigate mathematical properties of the problem domain — and are "not at all thinking on possible software"! This view of algorithmic requirements development has yet to gain general acceptance. We only look closer at one of several possible modelling cocnerns.

— Base Model

The base model makes precise the intrinsic components and their attributes. We express this model in *VDM* ([1, 2]).

The equations define pointed complete partial orders (cpo's), ie. domains, of mathematical values: × denote Cartesian product; $A \overrightarrow{m} B$ denotes the cpo of finite, partial functions from A into B; and | denotes non-discriminate union. Selectors, s-, also comment on the purpose of a value component.

Next we tabularize some comparisons:

Representation $\mathcal{A}$	
Railway Schedule &c.	
$\mathcal{A}$	Algorithmic Development Method
1	Domain Equation: $SCH = S_{\overrightarrow{m}}(S_{\overrightarrow{m}}(T_{\overrightarrow{m}}(D_{\overrightarrow{m}}A)))$ De-Curried: $SCH = (S \times S \times T \times D)_{\overrightarrow{m}}A$

versus

Figure 1: Base Model: $\mathcal{A}$ Domain Equations

1.0	$rs:RS$	$= Time \times SCH \times G \times TS$	System
2.0	$sc:SCH$	$= S_{\overrightarrow{m}}(S_{\overrightarrow{m}}(T_{\overrightarrow{m}}(D_{\overrightarrow{m}}A)))$	Schedule
3.0	$g:G$	$= S_{\overrightarrow{m}}(S_{\overrightarrow{m}}\mathbb{N}_1{}^+)$	Tracks
4.0	$ts:TS$	$= T_{\overrightarrow{m}}(LOC \times KIN)$	Trains
5.0	$lo:LOC$	$= S \mid (S \times \mathbb{N}_1 \times S)$	Location
6.0	$k:KIN$	$= L \times V \times AD$	Geometry+Kinematics
7.0	$ve:V$	$= \mathbb{N}_0 \times \underline{s\text{-}max}:\mathbb{N}_1$	Velocity
8.0	$ad:AD$	$= \mathtt{INTG} \times \underline{s\text{-}max}:\mathtt{INTG}$	Acceleration
9.0	$Time$	$= Week \times Day \times Hour \times Min$	Time
10.0	$Week$	$= 1 \mid 2 \mid \ldots \mid 52$	Week
11.0	Day	$= 0 \mid 1 \mid \ldots \mid 6$	Weekday
12.0	$Hour$	$= 0 \mid 2 \mid \ldots \mid 23$	Hour
13.0	Min	$= 0 \mid 1 \mid \ldots \mid 59$	Minute
14.0	D,A	$= \ldots$	Departure,Arrival
15.0	$s:S$	$= \mathtt{TOKEN}$	Station
16.0	$t:T$	$= \mathtt{TOKEN}$	Train
17.0	$le:L$	$= \mathbb{N}_1$	Length

Representation $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Schedule given as a 5-ary Predicate: $SCH(S,S,T,D,A)$ with indicated argument types: $S \times S \times T \times D \times A \rightarrow \mathtt{BOOL}$ Functional constraints on arrivals could be expressed Horn-clause-wise as an integrity constraint: $error() \leftarrow SCH(S_1,S_2,T,D,A') \;\&\; SCH(S_1,S_2,T,D,A'') \;\&\; A' \neq A''$

If the schedule is irregular *SCH* could be given as as a collection of factual clauses: $SCH(s_1,s_2,t,d,a)$.

If the schedule is 'fairly' regular *SCH* could be given as clausal form rules of the principal form:

18.0 $SCH(S_1,S_2,T,D,A) \leftarrow$
.1 $TRAINS(S_1,S_2,T,FST,ITV,LST,LAG)$,
.2 $INTERVALS(FST,ITV,LST,D)$,
.3 $A = LAG + D$

where *INTERVALS* yields all the possible values for D according to a regular schedule.

Factual clauses can, of course, be derived from this rule form by means of partial deduction expanding the *INTERVALS* predicate.

Discussion: The $\mathcal{K}$ form is neutral and uncommitted wrt. look-up demands: the five arguments are on par. Contrast this with the $\mathcal{A}$ shown. It tends to favour look-up of A given the four other arguments. Access functions

for other look-up forms are tedious to specify and understand.

On the other hand the $\mathcal{A}$ form suggests an efficient elaboration into an indexed representation favouring concise expression of for example stations.

The $\mathcal{K}$ approach moreover naturally accomodates a travel scheduling facility, which pieces connections together and is expressed by a recursive predicate. Rescheduling can then be expressed by way of meta-reasoning on scheduler rules. Deductive database technology (see paper (2) of [6]) offers efficient implementations of the rule based forms.

— Other Requirements Models

A number of other requirements models need be established before we have captured all there is needed to know before a software specification can be made. We will not detail these here, but refer, instead to [3] for references, and just briefly name them.

The conceptual model extends the base model with functions and behaviour models.

The physical model embeds the conceptual model in some "reality": define interfaces to and between relevant environment components, expresses safety criticality criteria, models dependability and computer/human & human/computer (ie. operator & user) interfaces.

Model execution (& simulation) can be used to validate models for which no closed mathematical structure can be convincingly built.

Appendix A gives more details.

3.1.3 Requirements Capture

Based on the mathematical requirements models we can now extract those parts for which it is being decided that the software shall monitor and take control.

3.2 Software Development

We focus only on the algorithmic software development stages outlined below:

Abstract Specification: Based on the requirements capture an abstraction is made on functionalities and behaviour of the software. The abstraction may decide on process decomposition into a parallel and distributed computing system.

Properties that relate to requirements models, but which only transpire indirectly from the model-theoretic software abstraction are formally verified.

Design Refinement: Based on the abstract specification a sequence of steps now gradually introduce concrete data structures and operations — for example with a view toward efficiency.

Coding: Finally code is manually or (semi) automatically derived for some (parallel and) imperative programming language.

3.3 Conclusions wrt. $\mathcal{A}$

- All reasoning took place during requirements development and oftentimes in various encoded forms (viz. the base model). Requirements capture 'converting' any such reasoning into further encodings in preparation for software development.
- The requirements, and also the software specification, although not fully illustrated, are expressed by logic formula over interpreted (that is model-theoretic) objects.
- The transition from conceptual to physical requirements models, and the transitions from abstract software specification via decreasingly abstract, increasingly concrete designs to code epitomizes stepwise design
- — with one of the main aims of all these refinement steps being efficiency.
- The aim of implementation is typically arithmetically oriented explicit state-changing computations.
- The issue of partial evaluation was inherent in our discussion of section 2 and in our use of stepwise refinement —
- — and compilation.

4 Comparative Analysis

We now compare the two appraches — in Tables 1–7.

Table 1 $\mathcal{A}$	
Formal Software Specification Aspects: Aims	
$\mathcal{A}$	Algorithmic Development Method
1	Specification of I/O function in mathematical sense to be implemented.
2	Use of such data structures as sets, Cartesian products, tuples, maps.
3	Distinction between abstract specification and concrete programming language.

versus

Table 1 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Description and specification of real world relationships as assertions
2	Computation results understood as proofs corresponding to logical consequences of $\mathcal{K}$.
3	Dual view of logic as both specification and programming language.

In $\mathcal{A}$ specification one may use logics at the specification level *versus* as an object language in $\mathcal{K}$. In $\mathcal{A}$ we aim at efficiency through specialisation of the application at hand *versus* through general methods in $\mathcal{K}$.

Table 2 $\mathcal{A}$	
Formal Specification Aspects/Presumptions	
$\mathcal{A}$	Algorithmic Development Method
1	Program $\neq$ Specification
2	Specification refined in stages into Executable Programs, Specifications not necessarily Executable
3	Focus on implementing data structure specification, for example Stacks, Queues, etc.
4	Finally: Compilation

versus

Table 2 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Logic as Object laguage Program and Specification often confused or partially identified
2	Specification of Executable Specifications = Declarative Programming cf. Kowalski: Algorithms = Logic + Control
3	Focus on domain knowledge
4	Finally, efficiency through general methods such as: Constraint Satisfaction and Intelligent Backtracking

In $\mathcal{A}$ we speak of computation, while in $\mathcal{K}$ we speak of deduction, or possibly induction or abduction.

Table 3 $\mathcal{A}$	
Mathematical Form of Computed Object	
$\mathcal{A}$	Algorithmic Development Method
1	Functional Conception of Computation Stepwise Refined and eventually algorithmitised by means of imperative programming language constructs, Operations on Data Objects $\Longrightarrow$ Deterministic Compilation λ-Calculus reduction as Computational Basis
3	Specifications are often 'functional' (algebraic)

versus

Table 3 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Relational Conception of Computation $\Longrightarrow$ Links with Relational Database, Non-determinism ("Backtracking") (Quasi) Parallellism $\Longrightarrow$ Proof Rule Deduction (Resolution) Unification as Computational Basis
2	Derivation of answers from Assertions ($\models, \vdash$) $\Longrightarrow$ Deduction Abduction (Cause Analysis) Induction (Machine Learning)
3	Specifications often 'relational' structures stressing $(m : n)$ relationships.

In $\mathcal{A}$ development we stepwise refine by elaborating, at development time, operations and data structures, *versus* remaining faithful to the problem domain in $\mathcal{K}$, where $\mathcal{K}$ may refine by exception elaborations and terminology concept model elaborations.

Table 4 $\mathcal{A}$	
Towards Computational Efficiency	
$\mathcal{A}$	Algorithmic Development Method
1	Stepwise Development (Refinement) Specification $\rightsquigarrow \cdots \rightsquigarrow$ Programs Informally or (semi)automatically Stressing Data Abstraction and Applicative Forms, Operations on sets (an an example) become operations on lists.
2	Development of Iterative Formalisms from Recursive ones
3	Paradigmatic Systems, Examples:
4	Compiler Development from Formal Specification of Static & Dynamic Semantics

versus

Table 4 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Specifications are ideally executable Efficiency eg. by Partial Deduction Constraint Satisfaction Methods Special Unification Methods
2	Initially uncommitted choice between Top-Down (Backward) use of Rules (cf. Recursive Forms) versus Bottom-Up (Forward) use of Rules (cf. Iterative Forms) Commitment through Meta-interpretation choice
3	Paradigmatic Systems, Examples:
4	Deductive Databases = subset of Prolog's logic in which true declarativity and termination (decidability) is achievable.

To Table 5 one could add that surprisingly few logic programming systems offer recursively defined, compound or structured types.

Table 5 $\mathcal{A}$ (Data) Types	
$\mathcal{A}$	Algorithmic Development Method
1	Traditionally Types as Protection and choice of representation, (for example: integer and floating point)
2	Types as Structuring Mechanism: Polymorphism Abstract Data Types

versus

Table 5 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Types viewed as classification of 'real world' entities $\Longrightarrow$ Terminological Logics (Concept Logics) Syllogistic Forms Order-sorted Logics as Generalisation of Hierarchies

In $\mathcal{A}$ our units of specification are possibly non-determinate or under-specified functions *versus* rules and facts in $\mathcal{K}$. The composition principle in $\mathcal{A}$ is functional composition *versus* catenation of assertions and object classification with multiple inheritance — possibly including non-monotonicity.

Table 6 $\mathcal{A}$ Descriptional Structuring Mechanisms	
$\mathcal{A}$	Algorithmic Development Method
1	Finely grained requirements models ([3]): Base, Function, Behaviour; Environment & Interface, Safety Criticality, Dependability, CHI
2	Block Structures and Modules with Encapsulation Procedures

versus

Table 6 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Database Conceptual Models and Schemes Inheritance Classification Hierarchies The Rule Clauses as 'self-contained' Units of Specification
2	(Definite) Horn Clauses

Coping with Exceptions: An example: In our model of a train departure & arrival schedule it was assumed that it was based on some modularity, say, a weekly plan: working days & week-ends. No exceptions were made. We know that there are schedule changes during certain holidays and seasons (to with: christmas, new year, national memorial days, &c.). In the algorithmic approach we can "fix" this either by 'extending' the schedule by making it a composition of a regular schedule, as first shown, with an exception schedule:

$$\begin{array}{lll} 19.0 & SCH = REG \times EXC & \text{Schedule} \\ 20.0 & REG = S_{\overline{m}}(S_{\overline{m}}(T_{\overline{m}}(D_{\overline{m}}A))) & \text{Regular} \\ 21.0 & EXC = Time\ {}_{\overline{m}}(T\ {}_{\overline{m}}Attr) & \text{Exception} \\ 22.0 & Attr = \ldots & \text{Exception Attributes} \end{array}$$

or we may 'repair' the original schedule by completing the orginal schedule so that it always spans a full year:

$$23.0 \quad SCH = Time\ {}_{\overline{m}}\ (S_{\overline{m}}(S_{\overline{m}}(T_{\overline{m}}(D_{\overline{m}}A))))$$

In the knowledge engineering approach, appealing to techniques of non-monotonic logic, one is able to modify the regular schedule by "overrriding" with the more specific singularities, thus avoiding proliferation of exceptions into the regular schemes.

Table 7 $\mathcal{A}$ Domain Model Exceptions and Non-monotonicity	
$\mathcal{A}$	Algorithmic Development Method
2	Exceptions handled by Explicit Enumeration of Cases

versus

Table 7 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	"Negation as (finite) Failure" (SLDNF-Resolution)
2	Distinction between $\vdash \neg A$ and $\nvdash A$.

5 Towards a Unified View

From the above comparisons and reflections concerning the classical application programming approach ($\mathcal{A}$) and the knowledge based approach ($\mathcal{K}$) concludingly we distill the following points:

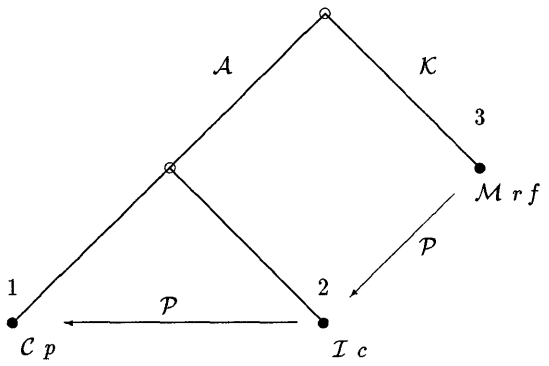
- The combining of methods from algorithmic and knowledge based approaches calls for language proposals which merge functional, imperative and logic programming paradigms, so as to overcome the functional/relational dichotomy.

  At present it seems unclear whether the object paradigm may assist in this merging or whether it presents an independent third approach yet to find its mathematical underpinnings.

- The declarative view of specifications/programs emphasised persistently in the logic oriented $\mathcal{K}$ approach suggests promotion of research in abstract interpretation, flow analysis, and partial evaluation (partial deduction) in the interest of automating efficient algorithmizations.

  An important result in this direction are the deductive data base systems, which represents an expedient unification of proper logical declarative languages and rather efficient algorithmitisations.

Figure 2: Partial Evaluation



5.1 A "Grand" Unifier

In section 2 we basically introduced the example now very informally diagrammed in figure 2.

Legend

- $\mathcal{A}$: algorithmic approach
- $\mathcal{K}$: knowledge engineering approach
- p: railway system program (to be compiled)
- c: railway system constant data structure (to be interpreted)
- f: knowledge base facts which reflect the railway system components
- r: knowledge base rules which reflect laws of railway systems
- $\mathcal{C}$: compiler
- $\mathcal{I}$: interpreter
- $\mathcal{M}$: inference machine
- $\mathcal{P}$: *Partial Evaluator* — a functional ([4]) which must satisfy the following laws:
  - $[\![\mathcal{M}]\!]_L\ r\ f = [\![\mathcal{I}]\!]_L\ c = [\![p]\!]_L = [\![\,[\![\mathcal{C}]\!]_{L'}\ p]\!]_M$
  - $([\![\mathcal{P}]\!]_{\mathcal{L}}\ \mathcal{M}\ r, f) \approx (\mathcal{I}, c)$
  - $[\![\mathcal{P}]\!]_{\mathcal{L}}\ \mathcal{I}\ d \approx p$

Here $[\![\ ]\!]$ denotes semantics brackets for languages $\mathcal{L}$, L, L' and (a machine lamguage) M — where typically $\mathcal{L}$, L and L' may be identical.

The laws express the following:

- From inference machine $\mathcal{M}$ and the rules and facts r, f partial evaluation $\mathcal{P}$ derives the interpreter $\mathcal{I}$ and the constant data structure c
- From the interpreter $\mathcal{I}$ and the constant data structure c partial evaluation $\mathcal{P}$ derives the compilable program p

You may think of $\mathcal{C}$, $\mathcal{I}$, $\mathcal{M}$ and $\mathcal{P}$ being written in the same language, typically LISP, and the representations of p, d, r and f being abstracted, for example as `S-expression`'s.

Nothing prevents f and c to be identically represented.

To find the un-typable functional $\mathcal{P}$ is a major research undertaking. To find efficient such partial evaluators, that is: functionals which *specialize* efficient interpreters & data structures and efficient programs is an even harder research problem. We think they are among the most important tasks of theoretical computer science!

6 The UNU/IIST

6.1 Overview

6.1.1 Aims & Objectives

UNU/IIST aims at assisting the developing world in meeting needs & strengthen capabilities in five activity areas: (i) deployment of advanced software, (ii) software technology management, (iii) development of own and exportable software, (iv) university education curriculum development and (v) participation in international research.

6.1.2 *UNU/IIST* Funding

UNU/IIST is the most recent Research & Training Centre (RTC) of the UNU to be established. It formally came into being on March 12th 1991, with the signing in Macau of agreements between the UNU, the Governor of Macau, and the governments of Portugal and the People's Republic of China. It is basically financed by an initial fund of US$20 million contributed to the *UNU/IIST* Endowment Fund, and it is pledged that this capital fund will be increased to US$30 million through contributions from other sources.

6.2 Programme Activities

UNU/IIST Progamme Activities center around projects, training, research, consultancy, dissemination and events.

Projects: *UNU/IIST* intends to engage in feasibility, demonstrator and technology transfer projects. All projects develop software using advanced techniques and are expected to last from 9 to 15 months. *UNU/IIST* staff, visiting experts and *project fellows* will conduct these Macau based projects, which are expected to be externally funded.

— **Feasibility** projects *formally*, but experimentally develop small, but difficult subsets of innovative software applications — and may lead to follow-on demonstrator projects. These projects may be prompted by, or lead to research done at *UNU/IIST* or within the *UNU/IIST Organic Network*.

— **Demonstrator** projects *rigorously* apply scalable, state-of-the-art techniques to applications that can serve as the basis for software development education courses — and may lead to follow-on technology transfer projects.

— **Technology Transfer** projects *systematically* develop core, prototype parts of planned products and shall lead to detailed plans and technical directions, which are then transferred to some developing world company for concluding, full-scale development.

The developed software will range from *reactive* (real-time, embedded) systems such as railway monitoring & control, river monitoring & flood control or cargo & customs clearance, via *analytic* systems such a traffic or crop fertilization planning, disease monitoring or disaster management, to *knowledge intensive* systems such as expert or knowledge based systems for decision support or university administration & management information.

Training: Courses & Seminars: *UNU/IIST* offers training through fellows participating in carefully supervised projects and through courses & seminars.

UNU/IIST will conduct three kinds of courses for *training participants* from the developing world.

Software usage: 2–4 week *training* — usually off-shore — courses will instruct software users and computer center operators to install & operate large scale software systems, and to prepare data for and evaluate results of their computations.

Software Technology Management: 2–4 week *awareness* — usually off-shore — courses will expose management in the intricasies of software technology management — in how to procure software, put out tender or bid for the development of software, and manage software development projects, software products and computing facilities. Common aspects include quality assurance, cost benefit & risk analysis, resource estimation, planning, allocation and scheduling, process modelling & simulation.

Development: 3 month *education* — Macau based — courses will teach software developers to develop application specific requirements, abstract & concrete program, and the engineering of large scale software systems: fit for use & purpose, correct, fault tolerant & safety critical, efficient, maintainable and portable. Subset education courses may be given — normally as off-shore — 2 week seminars.

Research: *UNU/IIST* will eventually embark on research in several areas. These include:

— **Programming Methodology:** We have focused in our earlier section on the *Programming* of software, where the objectives of programming were to insure correctness with respect to requirements, and efficiency &/or generality. *UNU/IIST* will initially research the area of *Duration Calculus* — we refer to [5].

— **Software Engineering:** In order to insure conformable, maintainable & portable software systems, software engineering employs such subsidiary techniques as ongoing — in the field — conformance testing, version control and configuration management, change request identification, monitoring & control, test case generation & validation, requirements & design decision tracking, and hypermedia supported documentation. *UNU/IIST* would like, through research, to better understand computable aspects of these techniques and their integration with programming.

— **Requirements Development:** Here we refer to the need to investigate mathematical techniques for relating informal narratives to base, functional and behavioral models, for relating the various requirements models and these to the requirements capture and stages of software development: programming & engineering.

UNU/IIST will especially emphasize both algorithmic and knowledge engineered requirements development. For the former approach see also appendix A.

— **Application Domain Modelling:** Requirements development, as above, need techniques. These are then to be applied to specific domain modelling.

In order for clients to procure software and expect delivery of what was intended, these customers must formulate their requirements relative to some 'normative' application domain narrative — which is furthermore supported by appropriate models.

In order for the software industry to be able to deliver trustworthy software products it must similarly be able to refer to, and rely on such 'standards' documents.

Consumers and producers must therefore undertake, for example through their business & industry associations, and most likely contracting private and public research institutes, to establish and later maintain such application area descriptions and models.

It is important that these descriptions and these models stay clear of unncessary design decisions, and, when un-avoidable, then to offer varieties of alternative such decisions.

We foresee, in this way, the establishment, of application domain "standards" for banking, insurance, various segments of transportation, similar varieties of production & manufacturing industries, &c. As it now is: everybody is re-establishing, mostly inside their own head, again and again, such understandings — and no progress is made. In the end: we get no closer to obtaining trustworthy systems!

We have often wondered about the almost total lack of problem (or application) domain recordings of the kind we here ask for.

The natural science fields of physics, chemistry, biology, etc. are doing exactly and basically only that! It is about time we also do it, wherever possible for the man-made universes of administration, business and industry!

Given that such domain recordings were expressed in either the Interpreted Algorithmic, or perhaps even better, but with less ease, in the Knowledge Based approach, and given the universal existence of a reasonably efficient partial evaluator $\mathcal{P}$ one can then very quickly, and at little cost, *specialize* any application domain to a compilable and efficiently executable program.

At *UNU/IIST* we expect to combine algorithmic requirements development and knowledge engineering methods to obtain such application domain models.

UNU/IIST will do some of the above mentioned research in Macau, and co-operate with other researchers around the globe. *UNU/IIST* will invite visiting experts and developing world research fellows to take part in this research. *UNU/IIST* is not presently expected to tackle the problem of determining a proper partial evaluator $\mathcal{P}$, let alone further understanding the relations between the $\mathcal{A}$ and the $\mathcal{K}$ approaches to application development.

6.3 A *UNU/IIST* Organic Network

An *Organic Network* will be linked to *UNU/IIST*. It will be an expanding circle of affiliated, co-operating software technology development centres, university computation science & engineering departments and research institutes. The network will focus on the developing world, but industrial world centres are expected to help secure the objectives of the network.

Aims of the network are to strengthen the identity, stability, quality and productivity, in developing world centres, with respect to development projects, university education and research in the areas covered by *UNU/IIST*.

Emphasis within the co-operation areas will be put on formulation of professional accreditation criteria including university curriculum development, affinity of software to its intended use — thereby helping to close the gap between consumer and producer, and correctness of software with respect to requirements definitions — thereby aiding the developing world in competitively producing highest quality software.

7 Conclusion

We have compared two approaches to software development and we have outlined areas of contemporary research related to their 'unification'. We have also outlined the *UNU/IIST strategy for prompting R&D in the area of software technology, including issues of education and international collaboration ... and future prospects of computer science* — as requested by the FGCS'92 organizers.

Finally we are left with the *issue ... of prompting creativity*. It is, of course, a crucial one; one that it might be difficult to convincingly argue is being sufficiently catered for. But we believe it will: using formal techniques supported by formally based tools, gets rid of most of the "grubby" work of keeping track of "zillions" of details. The *UNU/IIST* will demonstrate these techniques and tools to actual, and sometimes surprising applications across a very wide span. We believe that such *Master Class* tutoring will help foster the imagination. Now the human mind takes care of the rest: the creativity — we believe. That is: this is the best one can do: after proper formal education attend 'Master Classes' enabling the young researcher and developer to look over the shoulders of experienced creators.

8 Acknowledgements

The refined view of requirements development, in the special form presented here — and detailed in the appendix, is due to collaborative work with many colleagues in the *Basic Research Action* (BRA-3104) PROCOS: (*Provably Correct Systems*) project of the *Community of the European Countries'* (CEC's) *European Strategic Programme for Research in Information Technology* (ESPRIT).

References

[1] D. Bjørner. *Software Development. Volume I: Specification Principles — the VDM Approach.* Lecture Notes, Department of Computer Science, Technical University of Denmark, 710 pages, 1992.

[2] D. Bjørner. *Software Development. Volume II: Design Principles — the VDM Approach.* Lecture Notes, Department of Computer Science, Technical University of Denmark, 599 pages, Incomplete, 1992.

[3] D. Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia.* ACM Press, May 11–15 1992.

[4] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation*, Gl. Avernæs, Denmark, October 1987 1988. IFIP TC2 Working Conference, North-Holland Publ. Co., Amsterdam, The Netherlands.

[5] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

[6] J. Cohen, J. A. Robinson, J. Grant, J. Minker, Koichi Furukawa, and D. S. Warren. Special Section on Logic Programming: (1) *Logic and Logic Programming* (JAR, 40–65)), (2) *The Impact of Logic Programming on Databases* (JG&JM, 66–81), (3)*Logic Programming as the Integrator of the Fifth Generation Computer Systems Project* (KF, 82–92), (4) *Memoing for Logic Programming* (DSW, 93–112). *Communications of the ACM*, 35(3), March 1992.

[7] Anders P. Ravn et al., editor. *The ProCoS Project*, volume 5XX of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 1992.

[8] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kauffmann, 1987.

[9] J. W. Lloyd, editor. *Computational Logic; selected papers: (1) R.A. Kowalski:* Problems and promises of Computational Logic, *(2) A. Colmerauer:* An Introduction to Prolog III, *(3) K.R. Apt and D. Pedreschi:* Studies in pure Prolog: Termination, *(4) F. Baader et al.:* Concept Logics. ESPRIT Basic Research Action. Springer-Verlag, Heidelberg, Germany, 1990.

[1]

A Algorithmic Requirements Modelling

In the appendix we expand on the Algorithmic Requirements Modelling mentioned brifly in the paper.

The text is an extract of [3].

[1]The references pertaining to algorithmic development facets are rather limited to the (above-mentioned) ProCoS project.

A.1 Development Parts

To us the phases, stages & steps of development intertwine & iterate across two major phases:

1-3: *Requirements Development*

and:

4-6: *Software Development.*

These stages further decompose into stages:

1: Informal *Problem Domain Description,*

2: *Problem Domain Modelling,*

and:

3: *Requirements Capture.*

respectively:

4: *Abstract Software Design Specification,*

5: Steps of increasingly *Concrete Software Designs,*

and:

6: *Executable Code.*

Problem Domain Modelling (2) oftentimes consists of several, possibly concurrently "performed" steps. For embedded, real-time computing systems these steps include steps 2.1–2.4 below:

2.1: *base models,*

2.2: *functional laws,*

2.3: *behavioral laws,* and

2.4: *system architecture.*

The above steps may additionally encompass considerations of:

2.5: *safety criticality,*

2.6: *dependability,*

2.7: *performance,*

2.8: CHC/CHI,

(computer human communication/interfaces)

and also be subject to:

2.9: *model execution, ie. simulation.*

Thus 'concerns' 2.5–2.8 apply to steps 2.1–2.4, while simulation applies generally.

Each of the application domain modelling steps and concerns usually addresses distinct customer expectations (ie. client requirements (3)).

The iterative nature of development commonly dictates that stages (1–2–3–4) are interleaved, more specifically: that informal descriptions have to be enriched in preparation for subsequent steps of problem domain

modelling, capture and software architecture specification.

In the actual development, before a clarified picture can be given in terms of strict sequences of 1–2–3–4, we find that all of the above numbered steps and stages evolve in some manner that can perhaps best be described as a set of interacting co-routines.

Requirements capture (3) separates out from the various problem domain models those facets the software is expected to control.

A.2 Problem Domain Description

The purpose of developing an informal problem domain description is threefold: (i) to help extract what the problem is about from clients using some elicitation techniques, (ii) to make sure that the producer "also" understands the problem, and (iii) to serve, later, as part of the user documentation.

A.2.1 Synopsis

The synopsis gives a title to the project and a brief statement of purpose — usually couched in esoteric terms which, however, should lead the reader in the right direction.

A.2.2 Narrative

The narrative serves the rôle of establishing the terminology necessary to formulate expectations.

In any development project it is therefore important to start by establishing a *terminology* with a *taxonomy*, adhere to them, while critically reviewing and maintaining these. The narrative shall serve this point as well as providing an anchor for all the subsequent mathematical models. Their formal entities ("the x, y, z's of their formulae") must be "isomorphically" related to terms of the narrative.

For embedded, real-time computing systems it seems that the narrative should focus on *components* and their composition, *attributes* (component types and values), *events* (involving components and attributes), *procedures* (sequences of events sometimes emphasizing values), and *invariants* over components, values, events — especially typically concurrent event sequences, and hence procedures.

Formulating the narrative is an art: if not careful the narrator may inadvertently make undesired and even "hidden" design decisions.

We find that the narrative is best developed in steps of increasing concretisation.

A.2.3 Expectations

General, meta-, expectations are that the system shall *reflect* the components, *exhibit* the attributes, *handle* the events, *follow* the procedures, and *possess* the invariants. That is: state variables of the implementation mirrors "more or less isomorphically" the problem domain components, &c.

We 'believe', but this claim cannot yet be fully scientifically supported, that the above expectations also will lead to *adaptively maintainable* systems.

If not already captured in the narrative 'Expectations' capture desired (additional) properties.

A.3 Problem Domain Modelling

Problem domain modelling formalizes the narrative.

The purpose of application domain modelling is to secure a highest degree of confidence in one's understanding of the problem thereby insuring a best degree of affinity between the users' activity sphere and the software to be developed.

The problem domain modelling of the narrative and of these, and other, expectations, hence has as its *first* objective to allow formalization of expectations, as its *second* objective, to analyze feasibility of implied requirements, and, as its *third* objective, to synthesize specific monitoring & control, optimization planning, dependability, performance and computing system operator interfaces to problem domain, respectively computing equipment.

A.3.1 Base Model $\mathcal{M}$

Base modelling $\mathcal{M}$[2] was covered in subsection 3.1.2

A.3.2 Functional Laws $\mathcal{F}$

The purpose of establishing functional laws, $\mathcal{F}$, is to express what we might consider the most important *functional facts* of our problem domain, properties possessed independently of our possible computerisation. Typically, resulting software shall also possess these properties in some transparently encoded form, $\mathcal{F}_{\text{soft}}$.

A.3.3 Behavioral Laws $\mathcal{B}$

We shall use the term 'behavioral law' to cover a concept wider than that covered by 'control law'.

By a 'control', or 'behavioral law', $\mathcal{B}$, we mean a (mathematically expressable) principle which governs the possibly concurrent sequencing of events needed to fulfill basic *functional requirements*. Typically behavioral laws maintain functional laws while adhering to given procedures.

Etablishing behavioral laws amount to design decisions.

The purpose of identifying, among a set of alternatives, and specifying behavioral laws is to decompose behavior

[2]This and subsequent calligraphic letters are local to this appendix, that is: do not relate to those used elsewhere in the paper.

into two aspects: those controls to be performed overall by the computing system under development and those, detailed, controls to be done by other means — typically using conventional control-theoretic means.

A.3.4 System Architecture $\mathcal{Y}$

The purpose of establishing the 'systems architecture' and its laws, $\mathcal{Y}$, is to record the physical components needed to help obey the behavioral laws.

Choosing to express one (constituent) control principle over another (viz.: *Direct Digital Control (DDC), PID (proportion, integration and differentiation), stochastic, adaptive,* etc.) for individual components, and choosing one or more overall behavioral laws for sequencing operational phases (within which the former, constituent controls dominate), not only represents a choice, but implies insertion of new components into the problem domain.

These 'new' *plus* the 'old' components then represent the *Systems Architecture*, the *Design*. The new components are typically *AD/DA converters, clocks, sample and hold amplifiers, digital step or point set controllers,* as well as other *actuators and sensors*. Their composition, monitoring and control shall produce desired functional & behavioral requirements.

Thus the System Architecture is a final set of steps of narrative, base model and other problem domain models, which introduces the components particular to the computerisation. One should not forget here that software is among the components.

A.3.5 Safety Criticality $\mathcal{C}$

The purpose of establishing and expressing safety criticality criteria, $\mathcal{C}$ is to deal with those requirements that anticipate all the things that might go wrong while securing, given certain *assumptions*, that a minimum of functionality & behavior is maintained in the event of failures.

Thus safety predicates further constrain the functions denoted by the state variables — typically the traces of events that might occur.

A.3.6 Dependability $\mathcal{D}$

The purpose of dependability modelling is to calculate statistical measures for failure rates and compare them to requirements $\mathcal{D}$. Dependability requirements express probabilities of undesirable, but unavoidable behavior being below certain (acceptable) limits.

To perform dependability analysis we must have data on failure rates for non-software components.

Given a model that is believed to mirror proper assumptions one arrives at 'numbers' expressing total system failure rates in terms of a given system architecture. If these are acceptable, the architectural design can be approved. If they are not acceptable one must perform a redesign. The models give some hints as to what may constitute a proper, acceptable architecture.

Thus dependability requirements can usually be proposed, expressed, calculated and disposed of in this step of requirements development.

A.3.7 Performance $\mathcal{P}$

The purpose of performance modelling is to calculate statistical measures for performance and compare them to requirements $\mathcal{P}$.

Again the modelling is based on the base model architectures, at some level of abstraction. If the architectural design leads to acceptable performance characteristica, then the architecture can be approved, otherwise a redesign must take place.

Thus performance requirements can usually be proposed, expressed, calculated and, in principle, disposed of in this step of requirements development. Subsequent 'test measurements' may then validate the design.

A.3.8 CHC/CHI $\mathcal{I}$

The purpose of Computer-Human Communication Interface (CHC/CHI) modelling, $\mathcal{I}$, is to help design and validate interfaces between human operators and the computing system.

The idea is, however, that in a trustworthy design, one must establish models that portray human interaction psychologically, under stress, linguistically, including noisy misunderstanding, &c. Validated such models then contribute to the CHC/CHI requirements.

A.3.9 Model Execution

The purpose of simulation, ie. model execution, is to validate some of the models established in steps described earlier, and especially to ascertain some of the parameters or properties of these models, such which cannot be analytically justified, respectively calculated.

A.4 Requirements Capture $\mathcal{R}$

The purpose of requirements capture, $\mathcal{R}$, is to formally record those formal properties that the software shall possess — and is needed in order to secure affinity of resulting computing system to the application.

From the above we get:

24.0 $\quad \mathcal{R} \triangleq \mathcal{M} \wedge \mathcal{F} \wedge \mathcal{B} \wedge \mathcal{Y} \wedge \mathcal{C} \wedge \mathcal{D} \wedge \mathcal{P} \wedge \mathcal{I}$

The $\mathcal{D}$ and $\mathcal{P}$ terms can usually be verified, we conjecture[3], already at this stage wrt. the System Architecture (and its interface laws). Thus:

25.0 $\quad \mathcal{Y} \supset \mathcal{D} \wedge \mathcal{P}$

[3]This 'belief' is part of our future research plans.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

THE ROLE OF LOGIC IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE

J. A. Robinson

Syracuse University
New York 13244-2010, U.S.A.

ABSTRACT

The modern history of computing begins in the 1930s with the rigorous definition of computation introduced by Gödel, Church, Turing, and other logicians. The first universal digital computer was an abstract machine invented in 1936 by Turing as part of his solution of a problem in the foundations of mathematics. In the 1940s Turing's logical abstraction became a reality. Turing himself designed the ACE computer, and another logician-mathematician, von Neumann, headed the design teams which produced the EDVAC and the IAS computers. Computer science started in the 1950s as a discipline in its own right. Logic has always been the foundation of many of its branches: theory of computation, logical design, formal syntax and semantics of programming languages, compiler construction, disciplined programming, program proving, knowledge engineering, inductive learning, database theory, expert systems, theorem proving, logic programming and functional programming. Programming languages such as LISP and PROLOG are formal logics, slightly extended by suitable data structures and a few imperative constructs. Logic will always remain the principal foundation of computer science, but in the quest for artificial intelligence logic will be only one partner in a large consortium of necessary foundational disciplines, along with psychology, neuroscience, neurocomputation, and natural linguistics.

1 LOGIC AND COMPUTING

I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic. One could communicate with these machines in any language provided it was an exact language. In principle one should be able to communicate in any symbolic logic. *A. M. Turing, 1947*

The computer is the offspring of logic and technology. Its conception in the mid-1930s occurred in the course of the researches of three great logicians: Kurt Gödel, Alonzo Church, and Alan Turing, and its subsequent birth in the mid 1940s was largely due to Turing's practical genius and to the vision and intellectual power of another great logician-mathematician, John von Neumann. Turing and von Neumann played leading roles not only in the design and construction of the first computers but also in laying the general logical foundations for understanding the computation process and for developing computing formalisms.

Today, logic continues to be a fertile source of abstract ideas for novel computer architectures: inference machines, dataflow machines, database machines, rewriting machines. It provides a unified view of computer programming, (which is essentially a logical task) and a systematic framework for reasoning about programs. Logic has been important in the theory and design of high-level programming languages. Logical formalisms are the immediate models for two major logic programming language families: Church's lambda calculus for functional programming languages such as LISP, ML, LUCID and MIRANDA, and the Horn-clause-resolution predicate calculus for relational programming languages such as PROLOG, PARLOG, and GHC. Peter Landin noted over twenty years ago that ALGOL-like languages, too, were merely 'syntactically sugared' only-slightly-augmented versions of Church's lambda-calculus, and recently, another logical formalism, Martin-Löf's Intuitionistic Type Theory, has served (in, for example, Constable's NUPRL) as a very-high-level programming language, a notable feature of which is that a proof of a program's correctness is an automatic accompaniment of the program-writing process.

To design, understand and explain computers and programming languages; to compose and analyze programs and reason correctly and cogently about their properties; these are to practice an abstract logical art based upon (in H. A. Simon's apt phrase) a 'science of the artificial' which studies rational artifacts in abstraction from the engineering details of their physical realization, yet with an eye on their intrinsic efficiency. The formal logician has had to become also an abstract engineer.

1.1 LOGIC AND ARTIFICIAL INTELLIGENCE

Logic provides the vocabulary and many of the techniques needed both for analyzing the processes of representation and reasoning and for synthesizing machines that represent and reason. *N. J. Nilsson, 1991*

In artificial intelligence (AI) research, logic has been used (for example, by McCarthy and Nilsson) as a rational model for knowledge representation and (for example by Plotkin and Muggleton) as a guide for the organization of machine inductive inference and learning. It has also been used (for example by Wos, Bledsoe and Stickel) as the theoretical basis for powerful automated deduction systems which have proved theorems of interest to professional mathematicians. Logic's roles in AI, however, have been more controversial than its roles in the theory and practice of computing. Until the difference (if any) between natural intelligence and artificial intelligence is better understood, and until more experiments have tested the claims both of logic's advocates and of logic's critics concerning its place in AI research, the controversies will continue.

2 LOGIC AND THE ORIGIN OF THE COMPUTER.

Logic's dominant role in the invention of the modern computer is not widely appreciated. The computer as we know it today was invented in 1936, an event triggered by an important logical discovery announced by Kurt Gödel in 1930. Gödel's discovery decisively affected the outcome of the so-called Hilbert Program. Hilbert's goal was to formalize all of mathematics and then give positive answers to three questions about the resulting formal system: is it consistent? is it complete? it is decidable? Gödel found that no sufficiently rich formal system of mathematics can be both consistent and complete. In proving this, Gödel invented, and used, a high-level symbolic programming language: the formalism of primitive recursive functions. As part of his proof, he composed an elegant modular functional program (a set of connected definitions of primitive recursive functions and predicates) which constituted a detailed computational presentation of the syntax of a formal system of number theory, with special emphasis on its inference rules and its notion of proof. This computational aspect of his work was auxiliary to his main result, but is enough to have established Gödel as the first serious programmer in the modern sense. Gödel's computational example inspired Alan Turing a few years later, in 1936, to find an explicit but abstract logical model not only of the computing process, but also of the computer itself. Using these as auxiliary theoretical concepts, Turing disposed of the third of Hilbert's questions by showing that the formal system of mathematics is not decidable. Although his original computer was only an abstract logical concept, during the following decade (1937–1946) Turing became a leader in the design, construction and operation of the first real computers.

The problem of answering Hilbert's third question was known as the Decision Problem. Turing interpreted it as the challenge either to give an algorithm which correctly decides, for all formal mathematical propositions A and B, whether B is formally provable from A, or to show that is there no such algorithm. Having first clearly characterized what an algorithm is, he found the answer: there is no such algorithm.

For our present purposes the vital part of Turing's result is his characterization of what counts as an algorithm. He based it on an analysis of what a 'computing agent' does when making a calculation according to a systematic procedure. He showed that, when boiled down to bare essentials, the activity of such an agent is nothing more than that of (as we would now say) a finite-state automaton which interacts, one at a time, with the finite-state cells comprising an infinite memory.

Turing's machines are plausible abstractions from real computers, which, for Turing as for everyone else in the mid-1930s, meant a person who computes. The abstract Turing machine is an idealized model of any possible computational scheme such a human worker could carry out. His great achievement was to show that some Turing machines are 'universal' in that they can exactly mimic the behavior of any Turing machine whatever. All that is needed is to place a

coded description of the given machine in the universal machine's memory together with a coded description of the given machine's initial memory contents. How Turing made use of this universal machine in answering Hilbert's third question is not relevant to our purpose here. The point is that his universal machines are the abstract prototypes of today's stored program general-purpose computers. The coded description of each particular machine is the program which causes the universal machine to act like that particular machine.

Abstract and purely logical as it is, Turing's work had an obvious technological interpretation. There is no need to build a separate machine for each computing task. One need build only one machine—a universal machine—and one can make it perform any conceivable computing task simply by writing a suitable program for it. Indeed Turing himself set out to build a universal machine.

He began his detailed planning in 1944, when he was still fully engaged in the wartime British code-breaking project at Bletchley Park, and when the war ended in 1945 he moved to the National Physical Laboratory to pursue his goal full time. His real motive was already to investigate the possibility of artificial intelligence, a possibility he had frequently discussed at Bletchley Park with Donald Michie, I. J. Good, and other colleagues. He wanted, as he put it, to build a brain. By 1946 Turing completed his design for the ACE computer, based on his abstract universal machine. In designing the ACE, he was able to draw on his expert knowledge of the sophisticated new electronic digital technology which had been used at Bletchley Park to build special-purpose code-breaking machines (such as the Colossus). In the event, the ACE would not be the first physical universal machine, for there were others who were after the same objective, and who beat NPL to it. Turing's 1936 idea had started others thinking. By 1945 there were several people planning to build a universal machine. One of these was John von Neumann.

Turing and von Neumann first met in 1935 when Turing was an unknown 23-year-old Cambridge graduate student. Von Neumann was already famous for his work in many scientific fields, including theoretical physics, logic and set theory, and several other important branches of mathematics. Ten years earlier, he had been one of the leading logicians working on Hilbert's Program, but after Gödel's discovery he suspended his specifically logical researches and turned his attention to physics and to mathematics proper. In 1930 he emigrated to Princeton, where he remained for the rest of his life.

Turing spent two years (from mid-1936 to mid-1938) in Princeton, obtaining a doctorate under Alonzo Church, who in 1936 had independently solved the Decision Problem. Church's method was quite different from Turing's and was not as intuitively convincing. During his stay in Princeton, Turing had many conversations with von Neumann, who was enthusiastic about Turing's work and offered him a job as his research assistant. Turing turned it down in order to resume his research career in Cambridge, but his universal machine had already become an important item in von Neumann's formidable intellectual armory. Then came the war. Both men were soon completely immersed in their absorbing and demanding wartime scientific work.

By 1943, von Neumann was deeply involved in many projects, a recurrent theme of which was his search for improved automatic aids to computation. In late 1944 he became a consultant to a University of Pennsylvania group, led by J. P. Eckert and J. W. Mauchly, which was then completing the construction of the ENIAC computer (which was programmable and electronic, but not universal, and its programs were not stored in the computer's memory). Although he was too late to influence the design of the ENIAC, von Neumann supervised the design of the Eckert-Mauchly group's second computer, the EDVAC. Most of his attention in this period was, however, focussed on designing and constructing his own much more powerful machine in Princeton – the Institute for Advanced Study (IAS) computer. The EDVAC and the IAS machine both exemplified the so-called von Neumann architecture, a key feature of which is the fact that instruction words are stored along with data in the memory of the computer, and are therefore modifiable just like data words, from which they are not intrinsically distinguished.

The IAS computer was a success. Many close copies were eventually built in the 1950s, both in US government laboratories (the AVIDAC at Argonne National Laboratory, the ILLIAC at the University of Illinois, the JOHNIAC at the Rand

Corporation, the MANIAC at the Los Alamos National Laboratory , the ORACLE at the Oak Ridge National Laboratory, and the ORDVAC at the Aberdeen Proving Grounds), and in foreign laboratories (the BESK in Stockholm, the BESM in Moscow, the DASK in Denmark, the PERM in Munich, the SILLIAC in Sydney, the SMIL in Lund, and the WEIZAC in Israel); and there were at least two commercial versions of it (the IBM 701 and the International Telemeter Corporation's TC-1).

The EDSAC, a British version of the EDVAC, was running in Cambridge by June 1949, the result of brilliantly fast construction work by M.V. Wilkes following his attendance at a 1946 EDVAC course. Turing's ACE project was, however, greatly slowed down by a combination of British civil-service foot-dragging and his own lack of administrative deviousness, not to mention his growing preoccupation with AI. In May 1948 Turing resigned from NPL in frustration and joined the small computer group at the University of Manchester, whose small but universal machine started useful operation the very next month and thus became the world's first working universal computer. All of Turing's AI experiments, and all of his computational work in developmental biology, took place on this machine and its successors, built by others but according to his own fundamental idea.

Von Neumann's style in expounding the design and operation of EDVAC and the IAS machine was to suppress engineering details and to work in terms of an abstract logical description. He discussed both its system architecture and the principles of its programming entirely in such abstract terms. We can today see that von Neumann and Turing were right in following the logical principle that precise engineering details are relatively unimportant in the essential problems of computer design and programming methodology. The ascendancy of logical abstraction over concrete realization has ever since been a guiding principle in computer science, which has kept itself organizationally almost entirely separate from electrical engineering. The reason it has been able to do this is that computation is primarily a logical concept, and only secondarily an engineering one. To compute is to engage in formal reasoning, according to certain formal symbolic rules, and it makes no logical difference how the formulas are physically represented, or how the logical transformations of them are physically realized.

Of course no one should underestimate the enormous importance of the role of engineering in the history of the computer. Turing and von Neumann did not. They themselves had a deep and quite expert interest in the very engineering details from which they were abstracting, but they knew that the logical role of computer science is best played in a separate theater.

3 LOGIC AND PROGRAMMING

Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics. *J. von Neumann and H. Goldstine, 1947*

Much emphasis was placed by both Turing and von Neumann, in their discussions of programming, on the two-dimensional notation known as the *flow-diagram*. This quickly became a standard logical tool of early programming, and it can still be a useful device in formal reasoning about computations. The later ideas of Hoare, Dijkstra, Floyd, and others on the logical principles of reasoning about programs were anticipated by both Turing (in his 1949 lecture *Checking a Large Routine*) and von Neumann (in the 1947 Report *Planning and Coding of Problems for an Electronic Computing Instrument*). They stressed that programming has both a static and a dynamic aspect. The static text of the program itself is essentially an expression in some formal system of logic: a syntactic structure whose properties can be analyzed by logical methods alone. The dynamic process of running the program is part of the semantic meaning of this static text.

3.1 AUTOMATIC PROGRAMMING

Turing's friend Christopher Strachey was an early advocate, around 1950, of using the computer itself to translate from high-level 'mathematical' descriptions into low-level 'machine-language' prescriptions. His idea was to try to liberate the programmer from concern with 'how' to compute so as to be able to concentrate on 'what' to compute: in short, to think and write programs in a more natural and human idiom. Ironically, Turing himself was not much interested in this idea, which he had already in 1947 pointed out as an 'obvious' one. In fact, he seems to have had a hacker's pride in his fluent machine-language virtuosity. He was able to think directly and easily in terms of bare bit patterns and of the

unorthodox number representations such as the Manchester computer's reverse (i.e., low-order digits first) base-32 notation for integers. In this attitude, he was only the first among many who have stayed aloof from higher-level programming languages and higher-level machine architectures, on the grounds that a real professional must be aware of and work closer to the actual realities of the machine. One senses this attitude, for example, throughout Donald Knuth's monumental treatise on the art of computer programming.

It was not until the late 1950s (when FORTRAN and LISP were introduced) that the precise sequential details of how arithmetical and logical expressions are scanned, parsed and evaluated could routinely be ignored by most programmers and left to the computer to work out. This advance brought an immense simplification of the programming task and a large increase in programmer productivity. There soon followed more ambitious language design projects such as the international ALGOL project, and the theory and practice of programming language design, together with the supporting software technology of interpreters and compilers, quickly became a major topic in computer science. The formal grammar used to define the syntax of ALGOL was not initially accompanied by an equally formal specification of its semantics; but this soon followed. Christopher Strachey and Dana Scott developed a formal 'denotational semantics' for programs, based on a rigorous mathematical interpretation of the previously uninterpreted, purely syntactical, lambda calculus of Church. It was, incidentally, a former student of Church, John Kemeny, who devised the enormously popular 'best-selling' programming language, BASIC.

3.2 DESCRIPTIVE AND IMPERATIVE ASPECTS

There are two sharply-contrasting approaches to programming and programming languages: the descriptive approach and the imperative approach.

The descriptive approach to programming focusses on the static aspect of a computing plan, namely on the denotative semantics of program expressions. It tries to see the entire program as a timeless mathematical specification which gives the program's output as an explicit function of its input (whence arises the term 'functional' programming). This approach requires the computer to do the work of constructing the described output automatically from the given input according to the given specifications, without any explicit direction from the programmer as to how to do it.

The imperative approach focusses on the dynamic aspect of the computing plan, namely on its operational semantics. An imperative program specifies, step by step, what the computer is to do, what its 'flow of control' is to be. In extreme cases, the nature of the outputs of an imperative program might be totally obscure. In such cases one must (virtually or actually) run the program in order to find out what it does, and try to guess the missing functional description of the output in terms of the input. Indeed it is necessary in general to 'flag' a control-flow program with comments and assertions, supplying this missing information, in order to make it possible to make sense of what the program is doing when it is running.

Although a purely static, functional program is relatively easy to understand and to prove correct, in general one may have little or no idea of the cost of running it, since that dynamic process is deliberately kept out of sight. On the other hand, although an operational program is relatively difficult to understand and prove correct, its more direct depiction of the actual process of computation makes an assessment of its efficency relatively straightforward. In practice, most commonly-used high-level programming languages—even LISP and PROLOG—have both functional and operational features. Good programming technique requires an understanding of both. Programs written in such languages are often neither wholly descriptive nor wholly imperative. Most programming experts, however, recommend caution and parsimony in the use of imperative constructs. Some even recommend complete abstention. Dijkstra's now-classic Letter to the Editor (of the Communications of the ACM), entitled 'GOTO considered harmful' is one of the earliest and best-known such injunctions.

These two kinds of programming were each represented in pure form from the beginning: Gödel's purely descriptive recursive function formalism and Turing's purely imperative notation for the state-transition programs of his machines.

3.3 LOGIC AND PROGRAMMING LANGUAGES

In the late 1950s at MIT John McCarthy and his group began to program their IBM 704 using symbolic logic directly. Their system, LISP, is the first major example of a logic programming language intended for actual use on a computer. It is essentially Church's lambda calculus, augmented by a simple recursive data structure (ordered pairs), the conditional expression, and an imperative 'sequential construct' for specifying a series of consecutive actions. In the early 1970s Robert Kowalski in Edinburgh and Alain Colmerauer in Marseille showed how to program with another only-slightly-augmented system of symbolic logic, namely the Horn-clause-resolution form of the predicate calculus. PROLOG is essentially this system of logic, augmented by a sequentializing notion for lists of goals and lists of clauses, a flow-of-control notion consisting of a systematic depth-first, back-tracking enumeration of all deductions permitted by the logic, and a few imperative commands (such as the 'cut'). PROLOG is implemented with great elegance and efficiency using ingenious techniques originated by David H. D. Warren. The principal virtue of logic programming in either LISP or PROLOG lies in the ease of writing programs, their intelligibility, and their amenability to metalinguistic reasoning. LISP and PROLOG are usually taken as paradigms of two distinct logic programming styles (functional programming and relational programming) which on closer examination turn out to be only two examples of a single style (deductive programming). The general idea of purely descriptive deductive programming is to construe computation as the systematic reduction of expressions to a normal form. In the case of pure LISP, this means essentially the persistent application of reduction rules for processing function calls (Church's beta-reduction rule), the conditional expression, and the data-structuring operations for ordered pairs. In the case of pure PROLOG, it means essentially the persistent application of the beta-reduction rule, the rule for the distributing AND through OR, the rule for eliminating existential quantifiers from conjunctions of equations, and the rules for simplifying expressions denoting sets. By merging these two formalisms one obtains a unified logical system in which both flavors of programming are available both separately and in combination with each other. My colleague Ernest Sibert and I some years ago implemented an experimental language based on this idea (we called it LOGLISP). Currently we are working on another one, called SUPER, which is meant to illustrate how such reduction logics can be implemented naturally on massively parallel computers like the Connection Machine.

LISP, PROLOG and their cousins have thus demonstrated the possibility, indeed the practicality, of using systems of logic directly to program computers. Logic programming is more like the formulation of knowledge in a suitable form to be used as the axioms of automatic deductions by which the computer infers its answers to the user's queries. In this sense this style of programming is a bridge linking computation in general to AI systems in particular. Knowledge is kept deliberately apart (in a 'knowledge base') from the mechanisms which invoke and apply it. Robert Kowalski's well-known equational aphorism *'algorithm = logic + control '* neatly sums up the necessity to pay attention to both descriptive and imperative aspects of a program, while keeping them quite separate from each other so that each aspect can be modified as necessary in an intelligible and disciplined way.

The classic split between procedural and declarative knowledge again shows up here: some of the variants of PROLOG (the stream-parallel, committed-choice nondeterministic languages such as ICOT's GHC) are openly concerned more with the control of events, sequences and concurrencies than on the management of the deduction of answers to queries. The uneasiness caused by this split will remain until some way is found of smoothly blending procedural with declarative within a unified theory of computation.

Nevertheless, with the advent of logic programming in the wide sense, computer science has outgrown the idea that programs can only be the kind of action-plans required by Turing-von Neumann symbol-manipulating robots and their modern descendants. The emphasis is (for the programmer, but not yet for the machine designer) now no longer entirely on controlling the dynamic sequence of such a machine's actions, but increasingly on the static syntax and semantics of logical expressions, and on the corresponding mathematical structure of the data and other objects which are the denotations of the expressions. It is interesting to speculate how different the history of computing might have

been if in 1936 Turing had proposed a purely descriptive abstract universal machine rather than the purely imperative one that he actually did propose; or if, for example, Church had done so. We might well now have been talking of 'Church machines' instead of Turing machines.

We would be used to thinking of a Church machine as an automaton whose states are the expressions of some formal logic. Each of these expressions denotes some entity, and there is a semantic notion of *equivalence* among the expressions: equivalence means denoting the same entity. For example, the expressions

$$(23 + 4)/(13 - 4),\quad 1.3 + 1.7,\quad \lambda z.\ (2z + 1)^{1/2}\ (4)$$

are equivalent, because they all denote the number three. A Church machine computation is a sequence of its states, starting with some given state and then continuing according to the transition rules of the machine. If the sequence of states eventually reaches a terminal state, and (therefore) the computation stops, then that terminal state (expression) is the output of the machine for the initial state (expression) as input. In general the machine computes, for a given expression, another expression which is equivalent to it and which is as simple as possible. For example, the expression '3' is as simple as possible, and is equivalent to each of the above expressions, and so it would be the output of a computation starting with any of the expressions above. These simple-as-possible expressions are said to be in 'normal form'. The 'program' which determines the transitions of a Church machine through its successive states is a set of 'rewriting' rules together with a criterion for applying some one of them to any expression. A rewriting rule is given by two expressions, called the 'redex' and the 'contractum' of the rule, and applying it to an expression changes (rewrites) it to another expression. The new expression is a copy of the old one, except that the new expression contains an occurrence of the contractum in place of one of the occurrences of the redex.

If the initial state is (23 + 4)/(13 − 4) then the transitions are:

$$\begin{aligned} (23 + 4)/(13 - 4) &\text{ becomes } 27/(13 - 4),\\ 27/(13 - 4) &\text{ becomes } 27/9,\\ 27/9 &\text{ becomes } 3. \end{aligned}$$

Or if the initial state is $\lambda z.\ (2z + 1)^{1/2}\ (4)$, then the transitions are:

$$\begin{aligned} \lambda z.\ (2z + 1)^{1/2}\ (4) &\text{ becomes } ((2 \times 4) + 1)^{1/2}\\ ((2 \times 4) + 1)^{1/2} &\text{ becomes } (8 + 1)^{1/2}\\ (8 + 1)^{1/2} &\text{ becomes } 9^{1/2}\\ 9^{1/2} &\text{ becomes } 3. \end{aligned}$$

Most of us are trained in early life to act like a simple purely arithmetical Church machine. We all learn some form of numerical rewriting rules in elementary school, and use them throughout our lives (but of course Church's lambda notation is not taught in elementary school, or indeed at any time except when people later specialize in logic or mathematics; but it ought to be). Since we cannot literally store in our heads all of the infinitely many redex-contractum pairs <23 + 4, 27>, <2+2, 4> etc., infinite sets of these pairs are logically coded into simple finite algorithms. Each algorithm (for addition, subtraction, and so on) yields the contractum for any given redex of its particular type. We hinted earlier that an expression is in normal form if it is as simple as possible. To be sure, that is a common way to think of normal forms, and in many cases it fits the facts. Actually to be in normal form is not necessarily to be in as simple a form as possible. What counts as a normal form will depend on what the rewriting rules are. Normal form is a relative notion: given a set of rewriting rules, an expression in normal form is one which contains no redex.

In designing a Church machine care must be taken that no expression is the redex of more than one rule. The machine must also be given a criterion for deciding which rule to apply to an expression which contains distinct redexes, and also for deciding which occurrence of that rule's redexes to replace, in case there are two or more of them. A simple criterion is always to replace the leftmost redex occurring in the expression.

A Church machine, then, is a machine whose possible states are all the different expressions of some formal logic and which, when started in some state (i.e., when given some expression of that logic) will 'try' to compute its normal form. The computation may or may not terminate: this will depend on the rules and on the initial expression. Some of the expressions for some Church machines may have no normal form. Since for all interesting formal logics there are infinitely many expressions, a Church machine is not a finite-state automaton; so in practice the same provision must be made as in the case of the

Turing machines for adjoining as much external memory as needed during a computation.

Church machines can also serve as a simple model for parallel computation and parallel architectures. One has only to provide a criterion for replacing more than one redex at the same time. In Church's lambda calculus one of the rewriting rules ('beta reduction') is the logical version of executing a function call in a high-level programming language. Logic programming languages based on Horn-clause-resolution can also be implemented as Church machines, at least as far as their static aspects are concerned.

In the early 1960s Peter Landin, then Christopher Strachey's research assistant, undertook to convince computer scientists that not merely LISP, but also ALGOL, and indeed all past, present and future programming languages are essentially the abstract lambda calculus in one or another concrete manifestation. One need add only an abstract version of the 'state' of the computation process and the concept of 'jump' or change of state. Landin's abstract logical model combines declarative programming with procedural programming in an insightful and natural way.

Landin's thesis also had a computer-design aspect, in the form of his elegant abstract logic machine (the SECD machine) for executing lambda calculus programs. The SECD machine language is the lambda calculus itself: there is no question of 'compiling' programs into a lower-level language (but more recently Peter Henderson has described just such a lower-level SECD machine which executes compiled LISP expressions). Landin's SECD machine is a sophisticated Church machine which uses stacks to keep track of the syntactic structure of the expressions and of the location of the leftmost redex.

We must conclude that the descriptive and imperative views of computation are not incompatible with each other. Certainly both are necessary. There is no need for their mutual antipathy. It arises only because enthusiastic extremists on both sides sometimes claim that computing and programming are 'nothing but' the one or the other. The appropriate view is that in all computations we can expect to find both aspects, although in some cases one or the other aspect will dominate and the other may be present in only a minimal way. Even a pure functional program can be viewed as an implicit 'evaluate this expression and display the result' imperative (as in LISP's classic read-eval-print cycle).

4 LOGIC AND ARTIFICIAL INTELLIGENCE

In AI a controversy sprang up in the late 1960s over essentially this same issue. There was a spirited and enlightening debate over whether knowledge should be represented in procedural or declarative form. The procedural view was mainly associated with Marvin Minsky and his MIT group, represented by Hewitt's PLANNER system and Winograd's application of it to support a rudimentary natural language capability in his simple simulated robot SHRDLU. The declarative view was associated with Stanford's John McCarthy, and was represented by Green's QA3 system and by Kowalski's advocacy of Horn clauses as a logic-based deductive programming language. Kowalski was able to make the strong case that he did because of Colmerauer's development of PROLOG as a practical logic programming language. Eventually Kowalski found an elegant way to end the debate, by pointing out a procedural interpretation for the ostensibly purely declarative Horn clause sentences in logic programs.

There is an big epistemological and psychological difference between simply describing a thing and giving instructions for constructing it, which corresponds to the difference between descriptive and imperative programming. One cannot always see how to construct the denotation of an expression efficiently. For example, the meaning of the descriptive expression

> the smallest integer which is the sum of two cubes in two different ways.

seems quite clear. We certainly understand the expression, but those who don't already (probably from reading of Hardy's famous visit to Ramanujan in hospital) know that it denotes the integer 1729 will have to do some work to figure it out for themselves. It is easy to see that 1729 is the sum of two cubes in two different ways if one is shown the two equations

$$1729 = 1^3 + 12^3 \qquad 1729 = 10^3 + 9^3$$

but it needs at least a little work to find them oneself. Then to see that 1729 is the smallest integer with this property, one has to see somehow that all smaller integers lack it, and this means checking each one, either literally, or by some clever shortcut. To find 1729, in the first

place, as the denotation of the expression, one has to carry out the all of this work, in some form or another. There are of course many different ways to organize the task, some of which are much more efficient than others, some of which are less efficient, but more intelligible, than others. So to write a general computer program which would automatically and efficiently reduce the expression

> the smallest integer which is the sum of two cubes in two different ways

to the expression '1729' and equally well handle other similar expressions, is not at all a trivial task.

4.1 AI AND PROGRAMMING

Automatic programming has never really been that. It is no more than the automatic translation of one program into another. So there must be some kind of program (written by a human, presumably) which starts off the chain of translations. An assembler and a compiler both do the same kind of thing: each accepts as input a program written in one programming language and delivers as output a program written in another programming language, with the assurance that the two programs are equivalent in a suitable sense. The advantage of this technique is of course that the source program is usually more intelligible and easier to write than the target program, and the target program is usually more efficient than the source program because it is typically written in a lower-level language, closer to the realities of the machine which will do the ultimate work. The advent of such automatic translations opened up the design of programming languages to express 'big' ideas in a style 'more like mathematics' (as Christopher Strachey put it). These big ideas are then translated into smaller ideas more appropriate for machine languages. Let us hope that one day we can look back at all the paraphernalia of this program-translation technology, which is so large a part of today's computer science, and see that it was only an interim technology. There is no law of nature which says that machines and machine languages are intrinsically low-level. We must strive towards machines whose 'level' matches our own.

Turing and von Neumann both made important contributions to the beginnings of AI, although Turing's contribution is the better known. His 1950 essay *Computing Machinery and Intelligence* is surely the most quoted single item in the entire literature of AI, if only because it is the original source of the so-called Turing Test. The recent revival of interest in artificial neural models for AI applications recalls von Neumann's deep interest in computational neuroscience, a field he richly developed in his later years and which was absorbing all his prodigious intellectual energy during his final illness. When he died in early 1957 he left behind an uncompleted manuscript which was posthumously published as the book *The Computer and the Brain.*

4.2 LOGIC AND PSYCHOLOGY IN AI

> If a machine is to be able to learn something, it must first be able to be told it. *John McCarthy, 1957*

> I do not mean to say that there is anything wrong with logic; I only object to the assumption that ordinary reasoning is largely based on it. *M. L. Minsky, 1985*

AI has from the beginning been the arena for an uneasy coexistence between logic and psychology as its leading themes, as epitomized in the contrasting approaches to AI of John McCarthy and Marvin Minsky. McCarthy has maintained since 1957 that AI will come only when we learn how to write programs (as he put it) which have common sense and which can take advice. His putative AI system is a (presumably) very large knowledge base made up of declarative sentences written in some suitable logic (until quite recently he has taken this to be the first order predicate calculus), equipped with an inference engine which can automatically deduce logical consequences of this knowledge. Many well-known AI problems and ideas have arisen in pursuing this approach: the Frame Problem, Nonmonotonic Reasoning, the Combinatorial Explosion, and so on.

This approach demands a lot of work to be done on the epistemological problem of declaratively representing knowledge and on the logical problem of designing suitable inference engines. Today the latter field is one of the flourishing special subfields of AI. Mechanical theorem-proving and automated deduction have always been a source of interesting and hard problems. After over three decades of trying, we now have well-understood methods of systematic deduction which are of considerable use in practical applications.

Minsky maintains that humans rarely use logic in their actual thinking and problem solving, but adds that logic is not a good basis even for artificial problem solving—that computer programs based solely on McCarthy's logical deductive knowledge-base paradigm will fail to display intelligence because of their inevitable computational inefficiencies; that the predicate calculus is not adequate for the representation of most knowledge; and that the exponential complexity of predicate calculus proof procedures will always severely limit what inferences are possible.

Because it claims little or nothing, the view can hardly be refuted that humans undoubtedly are in some sense (biological) machines whose design, though largely hidden from us at present and obviously exceedingly complicated, calls for some finite arrangement of material components all built ultimately out of 'mere' atoms and molecules and obeying the laws of physics and chemistry. So there is an abstract design which, when physically implemented, produces (in ourselves, and the animals) intelligence. Intelligent machines can, then, be built. Indeed, they can, and do routinely, build and repair themselves, given a suitable environment in which to do so. Nature has already achieved NI—natural intelligence. Its many manifestations serve the AI research community as existence proofs that intelligence can occur in physical systems. Nature has already solved all the AI problems, by sophisticated schemes only a very few of which have yet been understood.

4.3 THE STRONG AI THESIS

> According to Strong AI, the computer is not merely a tool in the study of the mind; rather, the appropriately programmed computer really *is* a mind, in the sense that computers given the right programs can be literally said to *understand* and have other cognitive states.
>
> *J. R. Searle, 1980*

Turing believed, indeed was the first to propound, the Strong AI thesis that artificial intelligence can be achieved simply by appropriate programming of his universal computer. Turing's Test is simply a detection device, waiting for intelligence to occur in machines: if a machine is one day programmed to carry on fluent and intelligent-seeming conversations, will we not, argued Turing, have to agree that this intelligence, or at least this apparent intelligence, is a property of the program? What is the difference between apparent intelligence, and intelligence itself? The Strong AI thesis is also implicit in McCarthy's long-pursued project to reconstruct artificially something like human intelligence by implementing a suitable formal system. Thus the Turing Test might (on McCarthy's view) eventually be passed by a deductive knowledge base, containing a suitable repertory of linguistic and other everyday human knowledge, and an efficient and sophisticated inference engine. The system would certainly have to have a mastery of (both speaking and understanding) natural language. Also it would have to exhibit to a sufficent degree the phenomenon of 'learning' so as to be capable of augmenting and improving its knowledge base to keep it up-to-date both in the small (for example in dialog management) and in the large (for example in keeping up with the news and staying abreast of advances in scientific knowledge). In a recent vigorous defense of the Strong AI thesis, Lenat and Feigenbaum argued that if enough knowledge of the right kind is encoded in the system it will be able to 'take off' and autonomously acquire more through reading books and newspapers, watching TV, taking courses, and talking to people.

It is not the least of the attractions of the Strong AI thesis is that it is empirically testable. We shall know if someone succeeds in building a system of this kind: that indeed is what Turing's Test is for.

4.4 EXPERT SYSTEMS

Expert systems are limited-scale attempted practical applications of McCarthy's idea. Some of them (such as the Digital Equipment Corporation's system for configuring VAX computing systems, and the highly specialized medical diagnosis systems, such as MYCIN) have been quite useful in limited contexts, but there have not been as many of them as the more enthusiastic proponents of the idea might have wished. The well-known book by Feigenbaum & McCorduck on the Fifth Generation Project was a spirited attempt to stir up enthusiasm for Expert Systems and Knowledge Engineering in the United States by portraying ICOT's mission as a Japanese bid for leadership in this field.

There has indeed been much activity in devising specialized systems of applied logic whose axioms collectively represent a body of expert knowledge for some field (such as certain diseases, their symptoms and treatments) and whose deductions represent the process of solving problems posed

about that field (such as the problem of diagnosing the probable cause of given observed symptoms in a patient). This, and other, attempts to apply logical methods to problems which call for inference-making, have led to an extensive campaign of reassessment of the basic classicial logics as suitable tools for such a purpose. New, nonclassical logics have been proposed (fuzzy logic, probabilistic logic, temporal logic, various modal logics, logics of belief, logics for causal reationships, and so on) along with systematic methodologies for deploying them (truth maintenance, circumscription, non-monotonic reasoning, and so on). In the process, the notion of what is a logic has been stretched and modified in many different ways, and the current picture is one of busy experimentation with new ideas.

4.5 LOGIC AND NEUROCOMPUTATION

Von Neumann's view of AI was a 'logico-neural' version of the Strong AI thesis, and he acted on it with typical vigor and scientific virtuosity. He sought to formalize, in an abstract model, aspects of the actual structure and function of the brain and nervous system. In this he was consciously extending and improving the pioneer work of McCullogh and Pitts, who had described their model as 'a logical calculus immanent in nervous activity'. Here again, it was logic which served as at least an approximate model for a serious attack on an ostensibly nonlogical problem.

Von Neumann's logical study of self-reproduction as an abstract computational phenomenon was not so much an AI investigation as an essay in quasi-biological information processing. It was certainly a triumph of abstract logical formalization of an undeniably computational process. The self-reproduction method evolved by Nature, using the double helix structure of paired complementary coding sequences found in the DNA molecule, is a marvellous solution of the formal problem of self-reproduction. Von Neumann was not aware of the details of Nature's solution when he worked out his own logical, abstract version of it as a purely theoretical construction, shortly before Crick and Watson unravelled the structure of the DNA molecule in 1953. Turing, too, was working at the time of his death on another, closely-related problem of theoretical biology—morphogenesis—in which one must try to account theoretically for the unfolding of complex living structural organizations under the control of the programs coded in the genes. This is not exactly an AI problem. One cannot help wondering whether Turing may have been disappointed, at the end of his life, with his lack of progress towards realizing AI. If one excludes some necessary philosophical clarifications and preliminary methodological discussions, nothing had been achieved beyond his invention of the computer itself.

The empirical goal of finding out how the human mind actually works, and the theoretical goal of reproducing its essential features in a machine, are not much closer in the early 1990s than they were in the early 1950s. After forty years of hard work we have 'merely' produced some splendid tools and thoroughly explored plenty of blind alleys. We should not be surprised, or even disappointed. The problem is a very hard one. The same thing can be said about the search for controlled thermonuclear fusion, or for a cancer cure. Our present picture of the human mind is summed up in Minsky's recent book *The Society of Mind*, which offers a plausible general view of the mind's architecture, based on clues from the physiology of the human brain and nervous system, the computational patterns found useful for the organization of complex semantic information-processing systems, and the sort of insightful interpretation of observed human adult- and child-behavior which Freud and Piaget pioneered. Logic is given little or no role to play in Minsky's view of the mind.

Minsky rightly emphasizes (as logicians have long insisted) that the proper role of logic is in the context of justification rather than in the context of discovery. Newell, Simon and Shaw's 1956 well known propositional calculus theorem-proving program, the Logic Theorist, illustrates this distinction admirably. The Logic Theorist is a discovery simulator. The goal of their experiment was to make their program discover a proof (of a given propositional formula) by 'heuristic' means, reminiscent (they supposed) of the way a human would attack the same problem. As an algorithmic theorem-prover (one whose goal is to show formally, by any means, and presumably as efficiently as possible, that a given propositional formula is a theorem) their program performed nothing like as well as the best nonheuristic algorithms. The logician Hao Wang soon (1959) rather sharply pointed this out, but it seems that the psychologcial motivation of their investigation had eluded him (as indeed it has many others). They had themselves very much muddled the issue by contrasting their heuristic

theorem-proving method with the ridiculously inefficient, purely fictional, 'logical' one of enumerating all possible proofs in lexicographical order and waiting for the first one to turn up with the desired proposition as its conclusion. This presumably was a rhetorical flourish which got out of control. It strongly suggested that they believed it is more efficient to seek proofs heuristically, as in their program, than algorithmically with a guarantee of success. Indeed in the exuberance of their comparison they provocatively coined the wicked but amusing epithet 'British Museum algorithm' for this lexicogaphic-enumeration-of-all-proofs method—the intended sting in the epithet being that just as, given enough time, a systematic lexicographical enumeration of all possible texts will eventually succeed in listing any given text in the vast British Museum Library, so a logician, given enough time, will eventually succeed in proving any given provable proposition by proceeding along similar lines. Their implicit thesis was that a proof-finding algorithm which is guaranteed to succeed for any provable input is necessarily unintelligent. This may well be so: but that is not the same as saying that it is necessarily inefficient.

Interestingly enough, something like this thesis was anticipated by Turing in his 1947 lecture before the London Mathematical Society:

> ... if a machine is expected to be infallible, it cannot also be intelligent. There are several mathematical theorems which say almost exactly that.

5 CONCLUSION

Logic's abstract conceptual gift of the universal computer has needed to be changed remarkably little since 1936. Until very recently, all universal computers have been realizations of the same abstraction. Minor modifications and improvements have been made, the most striking one being internal memories organized into addressable cells, designed to be randomly accessible, rather than merely sequentially searchable (although external memories remain essentially sequential, requiring search). Other improvements consist largely of building into the finite hardware some of the functions which would otherwise have to be carried out by software (although in the recent RISC architectures this trend has actually been reversed). For over fifty years, successive models of the basic machine have been 'merely' faster, cheaper, physically smaller copies of the same device. In the past, then, computer science has pursued an essentially logical quest: to explore the Turing-von Neumann machine's unbounded possibilities. The technological challenge, of continuing to improve its physical realizations, has been largely left to the electrical engineers, who have performed miracles.

In the future, we must hope that the logician and the engineer will find it possible and natural to work more closely together to devise new kinds of higher-level computing machines which, by making programming easier and more natural, will help to bring artificial intelligence closer. That future has been under way for at least the past decade. Today we are already beginning to explore the possibilities of, for example, the Connection Machine, various kinds of neural network machines, and massively parallel machines for logical knowledge-processing.

It is this future that the bold and imaginative Fifth Generation Project has been all about. Japan's ten-year-long ICOT-based effort has stimulated (and indeed challenged) many other technologically advanced countries to undertake ambitious logic-based research projects in computer science. As a result of ICOT's international leadership and example, the computing world has been reminded not only of how central the role of logic has been in the past, as generation has followed generation in the modern history of computing, but also of how important a part it will surely play in the generations yet to come.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

PROGRAMS ARE PREDICATES

C.A.R. Hoare

Programming Research Group,
Oxford University Computing Laboratory,
11 Keble Road, Oxford, OX1 3QD, England.

Abstract

Requirements to be met by a new engineering product can be captured most directly by a logical predicate describing all its desired and permitted behaviours. The behaviour of a complex product can be described as the logical composition of predicates describing the behaviour of its simpler components. If the composition logically implies the original requirement, then the design will meet its specification. This implication can be mathematically proved before starting the implementation of the components. The same method can be repeated on the design of the components, until they are small enough to be directly implementable.

A programming language can be defined as a restricted subset of predicate notation, ensuring that the described behaviour may be efficiently realised by a combination of computer software and hardware. The restrictive notations give rise to a specialised mathematical theory, which is expressed as a collection of algebraic laws useful in the transformation and optimisation of designs. Non-determinism contributes both to reusability of design and to efficiency of implementation.

This philosophy is illustrated by application to hardware design, to procedural programs and to PROLOG. It is shown that the procedural reading of logic programs as predicates is different from the declarative reading, but just as logical.

1 Inspiration

It is a great honour for me to address this conference which celebrates the completion of the Fifth Generation Computer Systems project in Tokyo. I add my own congratulations to those of your many admirers and followers for the great advances and many achievements made by those who worked on the project. The project started with ambitious and noble goals, aiming not only at radical advances in Computer Technology, but also at the direction of that technology to the wider use and benefit of mankind. Many challenges remain; but the goal is one that inspires the best work of scientists and engineers throughout the ages.

For my part, I have been most inspired by the philosophy with which this project approaches the daunting task of writing programs for the new generation of computers and their users. I have long shared the view that the programming task should always begin with a clear and simple statement of requirements and objectives, which can be formalised as a specification of the purposes which the program is required to meet. Such specifications are predicates, with variables standing for values of direct or indirect observations that can be made of the behaviour of the program, including both questions and answers, input and output, stimulus and response. A predicate describes, in a neutral symmetric fashion, all permitted values which those variables may take when the program is executed. The over-riding requirement on a specification is clarity, achieved by a notation of the highest possible modularity and expressive power. If a specification does not *obviously* describe what is wanted, there is a grave danger that it describes what is *not* wanted; it can be difficult, expensive, and anyway mathematically impossible to check against this risk.

A minimum requirement on a specification language is that it should include in full generality the elementary connectives of Boolean Algebra: conjunction, disjunction, and negation — simple *and*, *or*, and *not*. Conjunction is needed to connect requirements, both of which must be met, for example,

- it must control pressure *and* temperature.

Disjunction is needed to allow tolerances in implementation

- it may deviate from optimum by one *or* two degrees.

And negation is needed for even more important reasons

- it must *not* explode!

As a consequence, it is possible to write a specification like

$$P \vee \neg P$$

which is always true, and so describes every possible observation of every possible product. Such a tolerant

specification is easy to satisfy, even by a program that gets into an infinite loop. In fact, such infinite failure will be treated as so serious that the tautologously true specification is the only one that it satisfies.

Another inspiring insight which I share with the Fifth Generation project is that programs too are predicates. When given an appropriate reading, a program describes all possible observations of its behaviour under execution, all possible answers that it can give to any possible question. This insight is one of the most convincing justifications for the selection of logic programming as the basic paradigm for the Fifth Generation project. But I believe that the insight is much more general, and can be applied to programs expressed in other languages, and indeed to engineering products described in any meaningful design notation whatsoever. It gives rise to a general philosophy of engineering, which I shall illustrate briefly in this talk by application to hardware design, to conventional sequential programs, and even to the procedural interpretation of PROLOG programs.

But it would be wholly invalid to claim that all predicates can be read as programs. Consider a simple but dramatic counter-example, the contradictory predicate

$$P \ \& \ \neg P$$

which is always false. No computer program (or anything else) can ever produce an answer which has a property P as well as its negation. So this predicate is not a program, and no processor could translate it into one which gives an answer with this self-contradictory property. Any theory which ascribes to an implementable program a behaviour which is known to be unimplementable must itself be incorrect.

A programming language can therefore be identified with only a subset of the predicates of predicate calculus; each predicate in this subset is a precise description of all possible behaviours of some program expressible in the language. The subset is designed to exclude contradictions and all other unimplementable predicates; and the notations of the language are carefully restricted to maintain this exclusion. For example, predicates in PROLOG are restricted to those which are definable by Horn clauses; and in conventional languages, the restrictions are even more severe. In principle, these gross restrictions in expressive power make a programming language less suitable as a notation for describing requirements in a modular fashion at an appropriately high level of abstraction.

The gap between a specification language and a programming language is one that must be bridged by the skill of the programmer. Given specification S, the task is to find a program P which satisfies it, in the sense that every possible observation of every possible behaviour of the program P will be among the behaviours described by (and therefore permitted by) the specification S. In logic, this can be assured with mathematical certainty by a proof of the simple implication

$$\vdash P \Rightarrow S.$$

A simple explanation of what it means for a program to meet its specification is one of the main reasons for interpreting both programs and specifications within the predicate calculus.

Now we can explain the necessity of excluding the contradictory predicate *false* from a programming notation. It is a theorem of elementary logic that

$$\vdash \mathit{false} \Rightarrow S,$$

so *false* enjoys the miraculous property of satisfying every specification whatsoever. Such miracles do not exist; which is fortunate, because if they did we would never need anything else, certainly not programs nor programming languages nor computers nor fifth generation computer projects.

2 Examples

A very simple example of this philosophy is taken from the realm of procedural programming. Here the most important observable values are those which are observed before the program starts and those which are observed after the program is finished. Let us use the variable x to denote the initial value and let x' be the final value of an integer variable, the only one that need concern us now. Let the specification say that the value of the variable must be increased

$$S = (x' > x)$$

Let the program add one to x

$$P = (x := x + 1)$$

The behavioural reading of this program as a predicate describing its effect is

$$P = (x' = x + 1)$$

i.e., the final value of x is one more than its initial value.

Every observation of the behaviour of P in any possible initial state x will satisfy this predicate. Consequently the Validity of the implication

$$\vdash P \Rightarrow S$$

i.e., $$\vdash x' = x + 1 \Rightarrow x' > x$$

will ensure that P correctly meets its specification. So does the program

$$x := x + 7,$$

but not

$$x := 2 \times x.$$

To illustrate the generality of my philosophy, my next examples will be drawn from the design of combinational hardware circuits. These can also be interpreted as predicates. A conventional *and-gate* with two input wires named a and b and a single output wire named x is described by a simple equation

$$x = a \wedge b.$$

The values of the three free variables are observed as voltages on the named wires at the end of a particular cycle of operation. At that time, the voltage on the output wire x is the lesser of the voltages on the input wires a and b. Similarly, an *or-gate* can be described by a different predicate with different wires

$$d = y \vee c,$$

i.e., the voltage on d is the greater of those on y and c. A simple wire is a device that maintains the same voltage at each of its ends, for example

$$x = y.$$

Now consider an assembly of two components operating in parallel, for example the and-gate together with the or-gate. The two predicates describing the two components have no variables in common; this reflects the fact that there is absolutely no connection between them. Consequently, their simultaneous joint behaviour consists solely of their two independent behaviours, and is correctly described by just the conjunction of the predicates describing their separate behaviours

$$(x = a \wedge b) \ \& \ (d = y \vee c)$$

This simple example is a convincing illustration of the principle that parallel composition of components is nothing but conjunction of their predicates, at least in the case when there is no possibility of interaction between them.

The principle often remains valid when the components are connected by variables which they share. For example, the wire which connects x with y can be added to the circuit, giving a triple conjunction

$$(x = a \wedge b) \ \& \ (x = y) \ \& \ (d = (y \vee c)).$$

This still accurately describes the behaviour of the whole assembly. The predicate is mathematically equivalent to

$$(d = (a \wedge b) \vee c) \ \& \ (x = y = (a \wedge b)).$$

When components are connected together in this way by the sharing of variable names (x and y), the values of the shared variables are usually of no concern or interest to the user of the product, and even the option of observing them is removed by enclosure, as it were, in a black box. The variables therefore need to be hidden or removed or abstracted from the predicate describing the observable behaviour of the assembly; and the standard way of eliminating free variables in the predicate calculus is by quantification.

In the case of engineering designs, existential quantification is the right choice. It is necessary that there exist an observable value for the hidden variable; but no one cares exactly what value it is. A formal justification is as follows. Let S be the specification for the program P, and let x be the variable to be hidden in P. Clearly, one could never wish to hide a variable which is mentioned in the specification, so clearly x will not occur free in S. Now the designer's original proof obligation without hiding is

$$\vdash P \Rightarrow S;$$

and the proof obligation after hiding is

$$\vdash (\exists x.P) \Rightarrow S.$$

By the predicate calculus, since x does not occur in S, these two proof obligations are the same.

But often quantification simplifies, as in our hardware example, where the formula

$$\exists x, y. \quad x = a \wedge b \ \& \ y = x \ \& \ d = y \vee c,$$

reduces to just

$$d = (a \wedge b) \vee c.$$

This mentions only the visible external wires of the circuit, and probably expresses the intended specification of the little assembly.

Unfortunately, not all conjunctions of predicates lead to implementable designs. Consider for example the conjunction of a negation circuit ($y = \neg x$) with the wire ($y = x$), connecting its output back to its input. In practice, this assembly leads to something like an electrical short circuit, which is completely useless — or even worse than useless, because it will prevent proper operation of any other circuit in its vicinity. So there is no specification (other than the trivial specification *true*) which a short-circuited design can reasonably satisfy. But in our oversimplified theory, the predicted effect is exactly the opposite. The predicate describing the behaviour of the circuit is a self-contradiction, equivalent to *false*, which is necessarily unimplementable.

One common solution to the problem is to place careful restrictions on the ways in which components can be combined in parallel by conjunction. For example, in combinational circuit design, it is usual to make a rigid distinction between input wires (like a or c) and output wires (like x or d). When two circuits are combined, the output wires of the first of them are allowed to be connected to the input wires of the second, but never

the other way round. This restriction is the very one that turns a parallel composition into one of its least interesting special cases, namely sequential composition. This means that the computation of the outputs of the second component has to be delayed until completion of the computation of the outputs of the first component.

Another solution is to introduce sufficient new values and variables into the theory to ensure that one can describe all possible ways in which an actual product or assembly can go wrong. In the example of circuits, this requires at least a three-valued logic: in addition to high voltage and low voltage, we introduce an additional value (written $\perp$, and pronounced "bottom"), which is observed on a wire that is connected simultaneously both to high voltage and to low voltage, i.e., a short circuit. We define the result of any operation on $\perp$ to give the answer $\perp$. Now we can solve the problem of the circuit with feedback, specified by the conjunction

$$x = \neg y \ \ \& \ \ y = x$$

In three-valued logic, this is no longer a falsehood: in fact it correctly implies that both the wires x and y are short circuited

$$x = y = \perp.$$

The moral of this example is that predicates describing the behaviour of a design must also be capable of describing all the ways in which the design may go wrong. It is only a theory which correctly models the possibility of error that can offer any assistance in avoiding it.

If parallelism is conjunction of predicates, disjunction is equally simply explained as introducing non-determinism into specifications, designs and implementations. If P and Q are predicates, their disjunction $(P \vee Q)$ describes a product that may behave as P or as Q, but does not determine which it shall be. Consequently, you cannot control or predict the result. If you want $(P \vee Q)$ to satisfy a specification S, it is necessary (and sufficient) to prove both that P satisfies S and that Q satisfies S. This is exactly the defining principle of disjunction in the predicate calculus: it is the least upper bound of the implication ordering. This single principle encapsulates all you will ever need to know about the traditionally vexatious topic of non-determinism. For example, it follows from this principle that non-deterministic specifications are in general easier to implement, because they offer a range of options; but non-deterministic implementations are more difficult to use, because they meet only weaker specifications.

Apart from conjunction (which can under certain restrictions be implemented by parallelism), and disjunction (which permits non-deterministic implementation), the remaining important operator of the predicate calculus is negation. What does that correspond to in programming? The answer is: nothing! Arguments about computability show that it can never be implemented, because the complement of a recursively enumerable set is not in general recursively enumerable. A commonsense argument is equally persuasive. It would certainly be nice and easy to write a program that causes an explosion in the process which it is supposed to control. It would be nice to get a computer to execute the negation of this program, and so ensure that the explosion never occurs. Unfortunately and obviously this is impossible. Negation is obviously the right way to *specify* the absence of explosion, but it cannot be used in *implementation*. That is one of the main reasons why implementation is in principle more difficult than specification. Of course, negation can be used in certain parts of programs, for example, in Boolean expressions: but it can never be used to negate the program as a whole. We will see later that PROLOG negation is very different from the kind of Boolean negation used in specifications.

The most important feature of a programming language is recursion. It is only recursion (or iteration, which is a special case) that permits a program to be shorter than its execution trace. The behaviour of a program defined recursively can most simply be described by using recursion in the definition of the corresponding predicate. Let $P(X)$ be some predicate containing occurrences of a predicate variable X. Then X can be defined recursively by an equation stating that X is a fixed point of P

$$X \stackrel{\text{def}}{=} P(X).$$

But this definition is meaningful only if the equation has a solution; this is guaranteed by the famous Tarski theorem, provided that $P(X)$ is a monotonic function of the predicate variable X. Fortunately, this fact is guaranteed in any programming language which avoids non-monotonic operators like negation. If there is more than one solution to the defining equation, we need to specify which one we want; and the answer is that we want the weakest solution, the one that is easiest to implement. (Technically, I have assumed that the predicate calculus is a complete lattice: to achieve this I need to embed it into set theory in the obvious way .)

The most characteristic feature of computer programs in almost any language is sequential composition. If P and Q are programs, the notation (P,Q) stands for a program which starts like P; but when P terminates, it applies Q to the results produced by P. In a conventional programming language, this is easily defined in predicate notation as relational composition, using conjunction followed by hiding in exactly the same way as our earlier combinational circuit. Let x stand for an observation of the initial state of all variables of a program, and let x' stand for the final state. Either or both of these may take the special value $\perp$, standing for non-termination or infinite failure, which is one of the worst ways in which a program can go wrong. Each program is a predicate $P(x,x')$ or $Q(x,x')$, describing a relation

between the initial state x and the final state x'. For example, there is an identity program Π (a null operation), which terminates without making any change to its initial state. But it can do this only if it starts in a proper state, which is not already failed

$$\Pi \stackrel{\text{def}}{=} (x \neq \bot \Rightarrow x' = x).$$

Sequential composition of P and Q in a conventional language means that the initial state of Q is the same as the final state produced by P; however the value of this intermediate state passed from P to Q is hidden by existential quantification, so that the only remaining observable variables are the initial state of P and the final state of Q. More formally, the composition (P, Q) is a predicate with two free variables (x and x') which is defined in terms of P and Q, each of which are also predicates with two free variables

$$(P, Q)(x, x') \stackrel{\text{def}}{=} \exists y.\ \ P(x, y)\ \ \&\ \ Q(y, x').$$

Care must be taken in the definition of the programming language to ensure that sequential composition never becomes self-contradictory. A sufficient condition to achieve this is that when either x or x' take the failure value $\bot$, then the behaviour of the program is entirely unpredictable: anything whatsoever may happen. The condition may be formalised by the statement that for all predicates P which represent a program

$$\forall x'.P(\bot, x')$$

and

$$\forall x.\ \ P(x, \bot) \Rightarrow \forall x'.P(x, x').$$

The imposition of this condition does complicate the theory, and it requires the theorist to prove that all programs expressible in the notations of the programming language will satisfy it. For example, the null operation Π satisfies it; and for any two predicates P and Q which satisfy the condition, so does their sequential composition (P, Q), and their disjunction $P \vee Q$, and even their conjunction $(P \wedge Q)$, provided that they have no variables in common. As a consequence any program written only in these restricted notations will always satisfy the required conditions. Such programs can therefore never be equivalent to *false*, which certainly does *not* satisfy these conditions.

The only reason for undertaking all this work is to enable us to reason correctly about the properties of programs and the languages in which they are written. The simplest method of reasoning is by symbolic calculation using algebraic equations which have been proved correct in the theory. For example, to compose the null operation Π before or after a program P does not change P. Algebraically this is expressed in a law stating that Π is the unit of sequential composition

$$(P, \Pi) = P = (\Pi, P).$$

Also, composition is associative; to follow the pair of operations (P, Q) by R is the same as following P by the pair of operations (Q, R)

$$((P, Q), R) = (P, (Q, R)).$$

3 PROLOG

In its procedural reading, a PROLOG program also has an initial state and a result; and its behaviour can be described by a predicate defining the relation between these two. Of course this is quite different from the predicate associated with the logical reading. It will be more complicated and perhaps less attractive; but it will have the advantage of accurately describing the behaviour of a computer executing the program, while retaining the possibility of reasoning logically about its consequences.

The initial state of a PROLOG program is a substitution, which allocates to each relevant variable a symbolic expression standing for the most general form of value which that variable is known to take. Such a substitution is generally called θ. The result θ' of a PROLOG program differs from that of a conventional language. It is not a single substitution, but rather a *sequence* of answer substitutions, which may be delivered one after the other on request. For example, the familiar PROLOG program

$$\text{append}\ (X, Y, Z)$$

may be started in the state

$$Z = [1, 2].$$

It will then produce on demand a sequence of three answer states

$$\begin{array}{lllll} X & = & [\ ], & Y & = & [1,2] \\ X & = & [1], & Y & = & [2] \\ X & = & [1,2], & Y & = & [\ ]. \end{array}$$

Infinite failure is modelled as before by the special state $\bot$; when it occurs, it is always the last answer in the sequence. Finite failure is represented by the empty sequence [ ]; and the program NO is defined as one that always fails in this way

$$NO(\theta, \theta') \stackrel{\text{def}}{=} (\theta \neq \bot \Rightarrow \theta' = [\ ]).$$

The program that gives an affirmative answer is the program YES; but the answer it gives is no more than what is known already, packaged as a sequence with only one element

$$YES(\theta, \theta') \stackrel{\text{def}}{=} (\theta = \bot \Rightarrow \theta' = [\theta]).$$

A *guard* in PROLOG is a Boolean condition b applied to the initial state θ to give the answer YES or NO

$$b(\theta,\theta') \stackrel{\text{def}}{=} \theta' = [\theta] \ \& \ (b\theta)$$
$$\vee \ \theta' = [\ ] \ \& \ (\neg b\theta).$$

Examples of such conditions are VAR and NONVAR.

The effect of the PROLOG $or(P;Q)$ is obtained by just appending the sequence of answers provided by the second operand Q to the sequence provided by the first operand P; and each operand starts in the same initial state

$$(P;Q)(\theta,\theta') \stackrel{\text{def}}{=} \exists X,Y.\ P(\theta,X) \ \& \ Q(\theta,Y)$$
$$\& \ \text{append}(X,Y,\theta).$$

The definition of *append* is the same as usual, except for an additional clause which makes the result of infinite failure unpredictable

$$\text{append}([\bot],Y,Z)$$
$$\text{append}([\ ],Y,Y)$$
$$\text{append}([X|Xs],Y,[X|Zs])$$
$$:- \ \text{append}\ (Xs,Y,Zs).$$

In all good mathematical theories, every definition should be followed by a collection of theorems, describing useful properties of the newly defined concept. Since NO gives no answer, its addition to a list of answers supply by P can make no difference, so NO is the unit of PROLOG semicolon

$$NO;P = P = P;NO.$$

Similarly, the associative property of appending lifts to the composition of programs

$$(P;Q);R = P;(Q;R).$$

The PROLOG conjunction is very similar to sequential composition, modified systematically to deal with a sequence of results instead of a single one. Each result of the sequence X produced by the first argument P is taken as an initial state for an activation of the second argument Q; and all the sequences produced by Q are concatenated together to give the overall result of the composition

$$(P,Q)(\theta,\theta') \stackrel{\text{def}}{=} \exists X,Y. \quad P(\theta,X)$$
$$\& \quad \text{each}\ (X,Y)$$
$$\& \quad \text{concat}\ (Y,\theta')$$

where

$$\text{each}([\ ],[\ ])$$
$$\text{each}([X|Xs],[Y|Ys])$$
$$:- \ Q(X,Y) \ \& \ \text{each}\ (Xs,Ys)$$

and

$$\text{concat}([\ ],[\ ])$$
$$\text{concat}([X|Xs],Z)$$
$$:- \ \text{append}\ (X,Y,Z) \ \& \ \text{concat}(Xs,Y)$$

The idea is much simpler than its formal definition; its simplicity is revealed by the algebraic laws which can be derived from it. Like composition in a conventional language, it is associative and has a unit YES

$$P,(Q,R) = (P,Q),R$$

$$(YES,P) = P = (P,YES).$$

But if the first argument fails finitely, so does its composition with anything else

$$(NO,P) = NO.$$

However (P,NO) is unequal to NO, because P may fail infinitely; the converse law therefore has to be weakened to an implication

$$NO \Rightarrow (P,NO).$$

Finally, sequential composition distributes leftward through PROLOG disjunction

$$((P;Q),R) = (P,R);(Q,R).$$

But the complementary law of rightward distribution certainly does not hold. For example, let P always produce answer 1 and let Q always produce answer 2. When R produces many answers, $(R,(P;Q))$ produces answers

$$1,2,1,2,\ldots$$

whereas $(R,P);(R,Q)$ produces

$$1,1,1,\ldots,2,2,2\ldots.$$

Many of our algebraic laws describe the ways in which PROLOG disjunction and conjunction are similar to their logical reading in a Boolean algebra; and the absence of expected laws also shows clearly where the traditional logical reading diverges from the procedural one. It is the logical properties of the procedural reading that we are exploring now.

The acid test of our procedural semantics for PROLOG is its ability to deal with the non-logical features like the cut (!), which I will treat in a slightly simplified form. A program that has been cut can produce at most one result, namely the first result that it would have produced anyway

$$P!(\theta,\theta') \stackrel{\text{def}}{=} \exists X.\ P(\theta,X) \ \& \ \text{trunc}\ (X,\theta').$$

The truncation operation preserves both infinite and finite failure; and otherwise selects the first element of a sequence

$$\begin{aligned}&\text{trunc}([\bot], Y)\\&\text{trunc}([\ ], [\ ])\\&\text{trunc}([X|Xs], [X]).\end{aligned}$$

A program that already produces at most one result is unchanged when cut again

$$P!! = P!$$

If only one result is wanted from a composite program, then in many cases only one result is needed from its components

$$(P;Q)! = (P!;Q!)!$$
$$(P,Q)! = (P,Q!)!$$

Finally, YES and NO are unaffected by cutting

$$YES! = YES, \quad NO! = NO.$$

PROLOG negation is no more problematic than the cut. It turns a negative answer into a positive one, a non-negative answer into a negative one, and preserves infinite failure

$$\sim P(\theta, \theta') \stackrel{\text{def}}{=} \exists Y.\ P(\theta, Y)\ \&\ \text{neg}\ (Y, \theta')$$

where

$$\begin{aligned}&\text{neg}([\bot], Z)\\&\text{neg}([\ ], [\theta])\\&\text{neg}([X|Xs], [\ ]).\end{aligned}$$

The laws governing PROLOG negation of truth values are the same as those for Boolean negation

$$\sim YES = NO \quad \text{and} \quad \sim NO = YES.$$

The classical law of double negation has to be weakened to intuitionistic triple negation

$$\sim\sim\sim P \ = \ \sim P.$$

Since a negated program gives at most one answer, cutting it makes no difference

$$\sim P = \ \sim (P!) = (\sim P)!$$

Finally, there is an astonishing analogue of one of the familiar laws of de Morgan

$$\sim (P;Q) = (\sim P, \sim Q).$$

The right hand side is obviously much more efficient to compute, so this law could be very effective in optimisation. The dual law, however, does not hold.

A striking difference between PROLOG negation and Boolean negation is expressed in the law that the negation of an infinitely failing program also leads to infinite failure

$$\sim \text{true} \ = \ \text{true}.$$

This states that *true* is a fixed point of negation; since it is the weakest of all predicates, there can be no fixed point weaker than it

$$(\mu X.\sim X) \ = \ \text{true}.$$

This correctly predicts that a program which just calls its own negation recursively will fail to terminate.

That concludes my simple account of the basic structures of PROLOG. They are all deterministic in the sense that (in the absence of infinite failure) for any given initial substitution θ, there is exactly one answer sequence θ' that can be produced by the program. But the great advantage of reading programs as predicates is the simple way in which non-determinism can be introduced. For example, many researchers have proposed to improve the sequential *or* of PROLOG. One improvement is to make it commute like true disjunction, and another is to allow parallel execution of both operands, with arbitrary interleaving of their two results. These two advantages can be achieved by the definition

$$(P\|Q)(\theta, \theta') \stackrel{\text{def}}{=} \exists X, Y.\ \ P(\theta, X)\ \&\ Q(\theta, Y) \ \&\ \text{inter}\ (X, Y, \theta')$$

where the definition of interleaving is tedious but routine

$$\begin{aligned}&\text{inter}([\bot], Y, Z) \qquad\qquad \text{inter}(X, [\bot], Z)\\&\text{inter}([\ ], Y, Y) \qquad\qquad \text{inter}(X, [\ ], X)\\&\text{inter}([X|Xs], Y, [X|Z] \ :\!-\ \text{inter}(Xs, Y, Z)\\&\text{inter}(X, [Y|Ys], [Y|Z]) \ :\!-\ \text{inter}(X, Ys, Z).\end{aligned}$$

Because appending is just a special case of interleaving, we know

$$\text{append}(X, Y, Z) \Rightarrow \text{inter}\ (X, Y, Z).$$

Consequently, sequential *or* is just a special case of parallel *or*, and is always a valid implementation of it

$$(P;Q) \Rightarrow (P\|Q).$$

The left hand side of the implication is more deterministic than the right; it is easier to predict and to control; it meets every specification which the right hand side also meets, and maybe more. In short, sequential *or* is in all ways and in all circumstances better than the parallel *or* — in all ways except one: it may be slower to implement on a parallel machine. In principle non-determinism is demonic; it never makes programming easier, and its only possible advantage is an increase in performance. However, in many cases (including this one) non-determinism also simplifies specifications and

designs, and facilitates reasoning about them at higher levels of abstraction.

My final example is yet another kind of disjunction, one that is characteristic of a commit operation in a constraint language. The answers given are those of exactly one of the two alternatives, the selection being usually non-deterministic: the only exception is in the case when one of the operands fails finitely, in which case the other one is selected. So the only case when the answer is empty is when both operands give an empty answer

$$\begin{aligned}(P[\!]Q) \stackrel{\text{def}}{=} & ((\theta' = [\ ]) \ \&\ P \ \&\ Q) \\ & \vee ((\theta' \neq [\ ]) \ \&\ (P \vee Q)) \\ & \vee P(\theta, \bot) \ \vee \ Q(\theta, \bot).\end{aligned}$$

(The last two clauses are needed to satisfy the special conditions described earlier). The definition is almost identical to that of the alternative command in Communicating Sequential Processes, from which I have taken the notation. It permits an implementation which starts executing both P and Q in parallel, and selects the one which first comes up with an answer. If the first elements of P and Q are guards, this gives the effect of flat Guarded Horn Clauses.

4 Conclusion

In all branches of applied mathematics and engineering, solutions have to be expressed in notations more restricted than those in which the original problems were formulated, and those in which the solutions are calculated or proved correct. Indeed, that is the very nature of the problem of solving problems. For example, if the problem is

- Find the GCD of 3 and 4

a perfectly correct answer is the trivially easy one

- the GCD of 3 and 4;

but this does not satisfy the implicit requirement that the answer be expressed in a much more restricted notation, namely that of numerals.

The proponents of PROLOG have found an extremely ingenious technique to smooth (or maybe obscure) the sharpness of the distinction between notations used for specification and those used for implementation. They actually use the same PROLOG notation for both purposes, by simply giving it two different meanings: a declarative meaning for purposes of specification, and a procedural meaning for purposes of execution. In the case of each particular program the programmer's task is to ensure that these two readings are consistent. Perhaps my investigation of the logical properties of the procedural reading will assist in this task, or at least explain why it is such a difficult one.

Clearly, the task would be simpler in a language in which the logical and procedural readings are even closer than they are in PROLOG. This ideal has inspired many excellent proposals in the development of logic and constraint languages. The symmetric parallel version of disjunction is a good example. A successful result of this research is still an engineering compromise between the expressive power needed for simple and perspicuous specification, and operational orientation towards the technology needed for cost-effective implementation.

Such a compromise will (I hope) be acceptable and useful, as PROLOG already is, in a wide range of circumstances and applications. In the remaining cases, I would like to maintain as far as possible the inspiration of the Fifth Generation Computing project, and the benefits of a logical approach to programming. To achieve this, I would give greater freedom of expression to those engaged in formalisation of the specification of requirements, and greater freedom of choice to those engaged in the design of efficiently implementable programming languages. This can be achieved only by recognition of the essential dichotomy of the languages used for these two purposes. The dichotomy can be resolved by embedding both languages in the same mathematical theory, and using logical implication to establish correctness.

But what I have described is only the beginning,—nothing more than a vague pointer to a whole new direction and method of research into programming languages and programming methodology. If any of my audience is looking for a challenge to inspire the next ten years of research, may I suggest this one? If you respond to the challenge, the programming languages of the future will not only permit efficient parallel and even non-deterministic implementations; they will also help the analyst more simply to capture and formalise the requirements of clients and customers; and then help the programmer by systematic design methods to exercise inventive skills in meeting those requirements with high reliability and low cost. I hope I have explained to all of you why I think this is important and exciting. Thank you again for this opportunity to do so.

Acknowledgements

I am grateful to Mike Spivey, He Jifeng, Robin Milner, John Lloyd, and Alan Bundy for assistance in preparation of this address.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

PANEL:
A Springboard for Information Processing in the 21st Century

Chairman: Robert A. Kowalski

Imperial College of Science, Technology and Medicine
Department of Computing, 180 Queen's Gate, London SW7 2BZ, England
rak@doc.ic.ac.uk

In general terms, the question to be addressed by the panel is simply whether the Fifth Generation technologies, developed at ICOT and other centres throughout the world, will lead the development of information processing in the next century.

Considered in isolation, the most characteristic of these technologies are:

- knowledge information processing applications,
- concurrent and constraint logic programming languages, and
- parallel computer architectures.

But it is the integration of these technologies, using logic programming to implement applications, and using multiple instruction, multiple data (MIMD) parallelism to implement logic programming, which is the most distinguishing characteristic of the Fifth Generation Project.

To assess the future prospects of the Fifth Generation technologies, we need to consider the alternatives. Might multi-media, communications, or data processing, for example, be more characteristic than artificial intelligence of the applications of the future? Might object-orientation be more characteristic of the languages; and sequential, SIMD, MISD, or massively parallel connectionist computers be more typical of the computer architectures?

Certainly many of these technologies have been flourishing during the last few years. Old applications still seem to dominate computing, at the expense of new Artificial Intelligence applications. Object-orientation has emerged as an alternative language paradigm, apparently better suited than logic programming for upgrading existing imperative software. Both conventional and radically new connectionist architectures have made rapid progress, while effective MIMD architectures are only now beginning to appear.

But it may be wrong to think of these alternatives as competitors to the Fifth Generation technologies. Advanced database and data processing systems increasingly use Artificial Intelligence techniques for knowledge representation and reasoning. Increasingly many database and programming language systems have begun to combine features of object-orientation and logic programming. At the level of computer architectures too, there seems to be a growing consensus that connectionism complements symbolic processing, in the same way that sub-symbolic human perception complements higher-level human reasoning.

But, because it provides the crucial link between applications and computer architectures, it is with the future of computer languages that we must be most concerned.

The history of computer languages can be viewed as a slow, but steady evolution away from languages that reflect the structure and behaviour of machines to languages that more directly support human modes of communication. It is relevant to the prospects of logic programming in computing, therefore, that logic programming has begun to have a great influence, in recent years, on models of human languages and human reasoning outside computing. This influence includes contributions to the development of logic itself, to the development of "logic grammars" in computational linguistics, to the modelling of common sense and non-monotonic reasoning in cognitive science, and to the formalisation of legal language and legal reasoning. Thus, if computer languages in the future continue to become more like human languages, as they have in the past, then the future of logic programming in computing, and the future impact of the Fifth Generation technologies in general, must be assured.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Finding the Best Route for Logic Programming

Hervé Gallaire
GSI
25 Bd de l'Amiral Bruix, 75782 Paris Cedex 16 France
gallaire@gsi.fr

Abstract

The panel chairman has asked us to deal with two questions relating Logic Programming (LP) to computing. They have to do with whether LP is appropriate (the most appropriate?) as a springboard for computing as a whole in the 21st century, or whether it is so only for aspects (characteristics) of computing. I do not think that there is a definite answer to these questions until one discusses the perspective from which they are asked or from which their answer is to be given. In summary, we can be very positive that LP will play an important role, but only if it migrates into other leading environments.

1 Which Perspective To Look From

We are asked to talk about directions for the future, for research as well as for development. Clearly, for me, there will not be a Yes/No answer to the questions debated on this panel. I don't shy away, but at the same time there are too many real questions behind the ones we are asked to address. Thus, I will pick the one aspects I am mostly connected to: research on deductive databases and constraint languages, experience in commercial applications development and in building architectures for such commercial developments. Whether these different perspectives lead to a coherent picture is questionable.

If I ask the question relative to computing as a whole, I can ask it from the perspective of a researcher, from that of a manufacturer, from that of a buyer of such systems and ask whether LP is the pervasive paradigm that will be the underlying foundation of computing as a whole from each of these perspectives.

If I ask the question relative to the characteristics of computing, I can look at computing from the perspective of an end-user, of an application developer (in fact from many such applications, e.g. scientific, business, CAD, decision support, teaching, office support, ...), of a system developer, of a language developer, of an architecture engineer, of a tool developer (again there are many such tools, e.g. software engineering, application analyst, etc). I can even look at it from the perspective of research in each of the domains related to the perspectives just listed, for example a researcher in user interface systems, a researcher in software engineering, in database languages, in knowledge representation, etc.

But the picture is even more complicated than it appears here; indeed I can now add a further dimension to the idea of perspective elaborated upon here. Namely I can ask whether LP is to be seen as the "real thing" or whether it is to be an abstract model essentially. For example, ask whether it is a good encompassing model for all research aspects of computing, for some of them (the perspectives), whether it is a good abstract model for computations, for information systems, for business models, even if they do not appear in this form to their users, this being asked for each type of computation carried out in a computing system.

Looking at these questions is to study whether LP should be the basis of the view of the world as manipulated at each or some of the following levels: user's level, at system level, at application designer level, at research level, ... or whether it should only be a model of it, i.e. a model in which they basic problems of the world (at that level) are studied, and that the two would match only in some occasions.

2 Global Perspective

I think we have to recognise that the world is definitely never going to be a one level world (ie providing in hardware a direct implementation of the world view); second that the world view will be made of multiple views; third we have to accept that different views will need different tools to study a version of a problem at that level; and fourth that it may be appropriate to use abstractions to get the appropriate knowledge into play. Consequently, neither LP nor any other paradigm will be the underlying foundation for computing; it is very appropriate however, for each paradigm to ask what its limits are. This is what has been my understanding of most projects around LP in the past ten to fifteen years; trying several angles, pushing to the limits. Developing hardware for example is one such worthwhile effort.

3 Model and Research Perspective

As a model of computing, from a research perspective, LP will continue to develop as the major candidate for giving a "coherent" view of the world, a seamless integration of the different needs of a computing system for which it has given good models. To come to examples, I believe that LP has made major contributions in the following areas: rule-based programming, with particularly results on deductive databases, on problem solving and AI, specific logics for time and belief, solutions to problems dealing with negation, to those dealing with constraint programming and to those dealing with concurrent programming. It will continue to do so for quite some time. In some cases it will achieve a dominant position; in others it will not, even if it remains a useful formalism. In the directions of research at ECRC, we have not attempted to get such a unified framework, even though we have tried to use whatever was understood in one area of research into the others (eg, constraints and parallelism, constraints and negation, ..). LP will not achieve the status of being the unique encompassing model adopted by everyone. Indeed, there are theoretical reasons that have to do with equivalence results and the human intelligence which makes it very unlikely that a given formalism will be accepted as the unique formalism to study. Further, there is the fact that the more we study, the more likely it is that we have to invent formalisms at the right level of abstraction for the problems at hand. Mapping this to existing formalisms is often possible but cumbersome. This has the side advantage that formalisms evolve as they target new abstractions; LP has followed that path.

4 Commercial Perspective

As a tool for computing in general, from a business or manufacturer's point of view LP has not achieved the status that we believed it would. Logic has found, at best, some niches where it can be seen as a potential commercial player (there are many Prolog programs embedded in several CASE tools for example, natural language tools are another example). When it comes to the industrial or commercial world things are not so different from those in the academic or research world: the resistance to new ideas is strong too, although for different reasons. Research results being very often unconclusive when it comes to their actual relevance or benefits in practical terms, only little risk is taken. Fads play an important role in that world where technical matters are secondary to financial or management matters; the object technology is a fad, but fortunately it is more than that and will bring real benefits to those adopting it. We have not explained LP in terms as easy to understand as done in the object world (modularity, encapsulation in particular). The need to keep the continuity with the so-called legacy applications is perhaps even stronger than fads. To propose a new computing paradigm to affect the daily work of any of the professionals (whether they develop new code or use it) is a very risky task. C++ is a C based language; we have no equivalent to a Cobol based logic language. And still, C++ is not a pure object oriented language. The reason why the entity-relationship modeling technique (and research) is successful in the business place is that it has been seen as an extension of the current practices (Cobol, relational) not as a rupture with them. SQL is still far from incorporating extensions that LP can already provide but has not well explained: where are the industrial examples of the recursive rules expressed in LP? what is the benefit (cost, performance, ...) of stating business rules this way as opposed to programming; and without recursion, or with limited deductive capabilities, relational systems do without logic or just borrow from it; isn't logic too powerful a formalism for many cases? LP, just like AI based technology, has not been presented as an extension of existing engines; rather it has been seen as alternatives to existing solutions, not well integrated with them; it has suffered, like AI from that situation. Are there then new areas where LP can take a major share of the solution space? In the area of constraint languages, there is no true market yet for any such language; and the need for integration to the existing environments is of a rather different nature; it may be sufficient to provide interfaces rather than integration. A not to be overlooked problem, however is that when it is embedded in logic, constraint programming needs a complex engine, that of logic; when it is embedded in C, even if it is less powerful or if it takes more to develop it (hiding its logical basis in some sense), it will appear less risky to the industrial partners who will use or build it.

5 More Efforts Needed

Let me mention three areas where success can be reached, given the current results, but where more efforts are needed. Constraint based packages for different business domains, such as transportation, job shop scheduling, personnel assignment, etc will be winners in a competitive world; but pay attention to less ambitious solutions in more traditional languages. Case tools and repositories will use heavily logic based tools, particularly the deductive database technology, when we combine it with the object based technology for what each is good at. Third and perhaps more importantly, there is a big challenge to be won. My appreciation of computing evolution is as follows: there will be new paradigms in terms of "how to get work done by a computer"; this will revolve around some simple notions: applications and systems will be packaged as objects and run as distributed object systems, communicating through messages and events; forerunners of these technologies can

already be seen on the desktop (not as distributed tools), such as AppleEvents, OLE and VisualBasic, etc and also in new operating systems or layers just above them (e.g. Chorus, CORBA, NewVave, etc). Applications will be written in whatever formalism is most appropriate for the task they have to solve, provided they offer the right interface to the communication mechanism; these mechanisms will become standardised. I believe that concurrent and constraint logic languages have a very important role to play in expressing how to combine existing applications (objects, modules). If LP had indeed the role of a conductor for distributed and parallel applications, it would be very exciting; it is possible. Following a very similar analysis, I think that LP rules, particularly when they are declaratively used, are what's needed to express business rules relating and coordinating business objects as perceived by designers and users. To demonstrate these ideas, more people, knowledgeable in LP need to work on industrial strength products, packaging LP and its extensions and not selling raw engines; then there will be more industrial interest in logic, which in the longer term will trigger and guarantee adequate research levels. This is what I have always argued was needed to be done as a follow up of ECRC research, but I have not argued convincingly. This is what is being done with much enthusiasm by start ups around Prolog, by others around constraint languages, e.g. CHIP; this is what is being started by BULL on such a deductive and object oriented system. Much more risk taking is needed.

6 Conclusion

From the above discussion the reader may be left with a somewhat mixed impression as to the future of our field; this is certainly not the intent. The future is bright, provided we understand where it lies. LP will hold its rank in the research as well as in the professional worlds. The major efforts that the 1980's have seen in this domain have played an essential role in preparing this future. The Japanese results, as well as the results obtained in Europe and in the USA, are significant in terms of research and of potential industrial impact. The major efforts, particularly the most systematic one, namely the Fifth Generation Project, may have had goals either too ambitious or not thoroughly understood by many. If we understand where to act, then there is a commercial future for logic. At any rate, research remains a necessity in this area.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

The Role of Logic Programming in the 21st Century

Ross Overbeek

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, Illinois 60439
overbeek@mcs

1 The Changing Role

Logic programming currently plays a relatively minor role in scientific problem-solving. Whether this role will increase in the twenty-first century hinges on one question: When will there be a large and growing number of applications for which the best available software is based on logic programming? It is not enough that there exist a few small, peripheral applications successfully based on logic programming; the virtues of logic programming must be substantial enough to translate into superior applications programs.

1.1 Applications

Applications based on logic programming are starting to emerge in three distinct contexts:

1. Applications based on the expressive power of logic programming in the dialects that support backtracking and the use of constraints. These applications frequently center on small, but highly structured databases and benefit dramatically from the ability to develop prototypes rapidly.

2. Parallel applications in which the expressive power of the software environment is the key issue. These applications arise from issues of real-time control. As soon as the performance adequately supports the necessary response, the simplicity and elegance of the solution become most important. In this context, dialects of committed-choice logic programming have made valuable contributions.

3. Parallel applications in which performance is the dominant issue. In these applications, successful solutions have been developed in which the upper levels of the algorithm are all implemented in a committed-choice dialect, and the lower layers in C or Fortran.

What is striking about these three contexts is that they have not been successfully addressed within a unified framework. Indeed, we are still far from achieving such a framework.

1.2 Unification of Viewpoints

Will a successful unification based on logic programming and parallelism emerge as a dominant technology? Two distinct answers to this question arise:

No, the use of logic programming will expand within the distinct application areas that are emerging. Logic programming will play an expanding role in the area of information processing (based on complex databases), which will see explosive growth in the Unix/C, workstation, mass software, and networking markets. On the other hand, logic programming will play quite a different role in the context of parallel computation. While an integration of the two roles is theoretically achievable, in practice it will not occur.

Yes, a single technology will be adopted that is capable of reducing complexity. Developing such a technology is an extremely difficult task, and it is doubtful that integration could have proceeded substantially faster. Now, however, the fundamental insights required to achieve an integration are beginning to occur. The computational framework of the twenty-first century—a framework dominated by advanced automation, parallel applications, and distributed processing—must be based on a technology that allows simple software solutions.

I do not consider these viewpoints to be essentially contradictory; there is an element of truth in each. It seems clear to me that the development and adoption of an integrated solution must be guided by attempts to solve demanding applications requirements. In the short run, this will mean attempts to build systems upon existing, proven technology. The successful development of a unified computational framework based on logic programming will almost certainly not occur unless there is a short-term effort that develops successful applications for the current computing market. However, the complex automation applications that will characterize the next century simply cannot be adequately addressed from within a computational framework that fails to solve the needs of both distributed computation and knowledge information processing.

The continued development of logic programming will require a serious effort to produce new solutions to sig-

nificant applications—solutions that are better than any existing solutions. The logic programming community has viewed its central goal as the development of an elegant computational paradigm, along with a demonstration that such a paradigm could be realized. Relatively few individuals have taken seriously the task of demonstrating the superiority of the new technology in the context of applications development. It is now time to change this situation. The logic programming community must form relationships with the very best researchers in important application areas, and learn what is required to produce superior software in these areas.

2 My Own Experiences

Let me speak briefly about my own experiences in working on computational problems associated with the analysis of biological genomes. Certainly, the advances in molecular biology are leading to a wonderful opportunity for mankind. In particular, computer scientists can make a significant contribution to our understanding of the fundamental processes that sustain life. Molecular biology has also provided a framework for investigating the utility of technologies like logic programming and parallel processing.

I believe that the first successful integrated databases to support investigations of genomic data will be based on logic programming. The reason is that logic programming offers the ability to do rapid prototyping, to integrate database access with computation, and to handle complex data. Other approaches simply lack the capabilities required to develop successful genomic databases. Current work in Europe, Japan, and America on databases to maintain sequence data, mapping data, and metabolic data all convince me that the best systems will emerge from those groups who base their efforts on logic programming.

Now let me move to a second area—parallel processing. Only a very limited set of applications really requires the use of parallel processing; however, some of these applications are of major importance. As an example, let me cite a project in which I was involved. Our group at Argonne participated in a successful collaboration with Gary Olsen, a biologist at the University of Illinois, and with Hideo Matsuda of Kobe University. We were able to create a tool for inferring phylogenetic trees using a maximum likelihood algorithm, and to produce trees that were 30–40 times more complex than any reported in the literature. This work was done using the Intel Touchstone DELTA System, a massively parallel system containing 540 i860 nodes.

For a number of reasons, our original code was developed in C. We created a successful tool that exhibited the required performance on both uniprocessors and larger parallel systems. We find ourselves limited, however, because of load-balancing problems, which are difficult to address properly in the context of the tools we chose.

We are now rewriting the code using bilingual programming, with the upper levels coded in PCN (a language deriving much of its basic structure from committed-choice logic programming languages) and its lower levels in C. This approach will provide a framework for addressing the parallel processing issues in the most suitable manner, while allowing us to optimize the critical lower-level floating-point computations. This experience seems typical to me, and I believe that future systems will evolve to support this programming paradigm.

3 Summary

Balance between the short-term view and the longer-range issues relating to an adequate integration is necessary to achieve success for logic programming. The need to create an environment to support distributed applications will grow dramatically during the 1990s. Exactly when a solution will emerge is still in doubt; however, it does seem likely that such an environment will become a fundamental technology in the early twenty-first century. Whether logic programming plays a central role will depend critically on efforts in Japan, Europe, and America during this decade. If these efforts are not successful, less elegant solutions will become adopted and entrenched.

This issue represents both a grand challenge and a grand opportunity. Which approach will dominate in the next century has not yet been determined; only the significance of developing the appropriate technology is completely clear.

Acknowledgments

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Object-Based Versus Logic Programming

Peter Wegner

Brown University, Box 1910, Providence, RI. 02912

pw@cs.brown.edu

Abstract: This position paper argues that mainstream application programming in the 21st century will be object-based rather than logic-based for the following reasons. 1) Object-based programs model application domains more directly than logic programs. 2) Object-based programs have a more flexible program structure than logic programs. 3) Logic programs can be intractable, in part because the satisfiability problem is NP-complete. 4) Soundness limits the granularity of thinking while completeness limits its scope. 5) Inductive, abductive, probabilistic, and nonmonotonic reasoning sacrifice the certainty of deduction for greater heuristic effectiveness. 6) Extensions to deductive logic like nonmonotonic or probabilistic reasoning are better realized in a general computing environment than as extensions to logic programming languages. 7) Object-based systems are open in the sense of being both reactive and extensible, while logic programs are not reactive and have limited extensibility. 8) The don't-know nondeterminism of Prolog precludes reactiveness, while the don't-care nondeterminism of concurrent logic programs makes them nonlogical.

1. Modeling Power and Computability

Object-based programs model application domains more directly than logic programs. Computability is an inadequate measure of modeling capability since all programming languages are equivalent in their computing power. A finer (more discriminating) measure, called "modeling power", is proposed that is closely related to "expressive power", but singles out modeling as the specific form of expressiveness being studied. Features of object-based programming that contribute to its modeling power include:

- assignment and object identity

Objects have an identity that persists when their state changes. Objects with a mutable state capture the dynamically changing properties of real-world objects more directly than mathematical predicates of logic programs.

- data abstraction by information hiding

Objects specify the abstract properties of data by applicable operations without commitment to a data representation. Data abstraction is a more relevant form of abstraction for modeling than logical abstraction.

- messages and communication

Messages model communication among objects more effectively than logic variables. The mathematical behavior of individual objects can be captured by algebras or automoata, but communication and synchronization protocols actually used in practical object-based and concurrent systems have no neat mathematical models.

These features are singled out because they cannot be easily expressed by logic programs. Shapiro [Sh1] defines the comparative expressive power (modeling power) of two languages in terms of the difficulty of mapping programs of one language into the other. Language L1 is said to be more expressive than language L2 if programs of L2 can be easily mapped into those of L1 but the reverse mapping is difficult (according to a complexity metric for language mappings).

The specification of comparative expressive power in terms of mappings between languages is not entirely satisfactory. For example, mapping assembly languages into problem-oriented languages is difficult because of lack of design rather than quality of design. However, when applied to two well-structured language classes like object-based and logic languages this approach does appear promising.

Since logic programs have a procedural interpretation with goal atoms as procedure calls and logic variables as shared communication channels, logic programming can be viewed as a special (reductive) style of procedure-oriented programming. Though language features like nondeterminism, logic variables, and partially instantiated structures are not directly modeled, the basic structure of logic programs is procedural. In contrast, object-oriented programs in Smalltalk or C++ do not have a direct interpretation as logic programs, since objects and classes cannot be easily modeled. Computational objects that describe behavior by collections of operations sharing a hidden state cannot be easily mapped into logic program counterparts.

2. Limitations of Inference and Nondeterministic Control

All deduction follows from the principle that if an element belongs to a set then it belongs to any superset. The Aristotelian syllogism *"All humans are mortal, Socrates is human, therefore Socrates is mortal"* infers that Socrates belongs to the superset of mortals from the fact that he belongs to the subset of humans. This problem can be specified in Prolog as follows:

Prolog clause: *mortal(x)* ← *human(x).*
Prolog fact: *human(Socrates).*
Prolog goal: *mortal(Socrates).*

The clause *"mortal(x)* ← *human(x)"*, which specifies that the set of mortals is a superset of the set of humans, allows the goal *"mortal(Socrates)"* to be proved from the fact *"human(Socrates)"*.

A Prolog clause of the form *"P(x) if Q(x)"* asserts that the set of facts or objects satisfying Q is a subset of those satisfying P, being equivalent to the assertion *"For all x, Q(x) implies P(x)"*. A Prolog goal G(x) is true if there are facts in the data-

base that satisfy G by virtue of the set/subset relations implied by the clauses of the Prolog program. Prolog resolution and unification allows the subset of all database facts satisfying G to be found by set/subset inference.

Inferences of the form *"set(x) if subset(x)"* are surprisingly powerful, permitting all of mathematics to be expressed in terms of set theory. But the exclusive use of "set if subset" inference for computation and/or thinking is unduly constraining, since both computation and thinking go beyond mere classification. Thinking includes heuristic mechanisms like generalization and free association that go beyond deduction.

Nondeterminism is another powerful computation mechanism that limits the expressive power of logic programs. Prolog nondeterminically searches the complete goal tree for solutions that satisfy the goal. In a Prolog program with a predicate P appearing in the clause head of N clauses *"P(Ai) → Bi"*, a goal P(A) triggers nondeterministic execution of those bodies Bi for which A unifies with Ai. This execution rule can be specified by a choice statement of the form:

choice (A1|B1, A2|B2, ... , AN|BN) endchoice
nondeterministically execute the bodies Bi of all clauses for which the clause head P(Ai) unifies with the goal P(A).

Bodies Bi are guarded by patterns Ai that must unify with A for Bi to qualify for execution. This form of nondeterminism is called *don't-know nondeterminism* because the programmer need not predict which inference paths lead to successful inference of the goal. Prolog programs explore all alternatives until a successful inference path is found and report failure only if no inference path allows the goal to be inferred.

The order in which nondeterministic alternatives are explored is determined by the system rather than by the user, though the user can influence execution order by the order of listing alternatives. Depth-first search may cause unnecessary nonterminating computation, while breadth-first search avoids this problem but is usually less efficient. Prolog provides mechanisms like the cut which allows search mechanisms to be tampered with. This extra flexibility undermines the logical purity of Prolog programs.

Sequential implementation of don't-know nondeterminism requires backtracking from failed inference paths so that the effects of failed computations become unobservable. Since programs cannot commit to an observable output until a proof is complete, don't-know nondeterminism cannot be used as a computational model for *reactive systems* that respond to external stimuli and produce incremental output [Sh2].

3. Intactability and Satisfiability

Certain well-formulated problems like the halting problem for Turing machines are noncomputable. Practical computability is further restricted by the requirement of tractability. A problem is tractable if its computation time grows no worse than polynomially with its size and intractable if its computation time grows at least exponentially.

The class P of problems computable in polynomial time by a deterministic Turing machine is tractable, while the class NP of problems computable in polynomial time by a nondeterministic Turing machine has solutions checkable in polynomial time though it may take an exponential time to find them [GJ]. The question whether P = NP is open, but the current belief is that NP contains inherently intractable problems, like the *satisfiability problem*, that are not in P.

The satisfiability problem is NP-complete; a polynomial time algorithm for satisfiability would allow all problems in NP to be solved in polynomial time. The fundamental problem of theorem proving, that of finding whether a goal can be satisfied, is therefore intractable unless it turns out that P = NP.

The fact that satisfiability is intractable is not unacceptable especially when compared to the fact that computability is undecidable. But in practice exponential blowup arises more frequently in logic programming than undecidability arises in traditional programming. Sometimes the intractability is inherent in the sense that there is no tractable algorithm that solves the problem. But in many cases more careful analysis can yield a tractable algorithm. Consider for example the sorting problem which can be declaratively specified as the problem of finding an ordered permutation.

sort(x) :- permutation(x), ordered(x).

Direct execution of this specification requires n-factorial steps to sort n elements, while more careful anlysis yields algorithms like quicksort that require only n logn steps. High-level specifications of a problem by logic programs can lead to combinatorially intractable algorithms for problems that are combinatorially tractable when more carefully analyzed.

The complexity of logic problem solving is often combinatorially unacceptable even when problems do have a solution. The intractability of the satisfiability problem causes some problems in artificial intelligence to become intractable when blindly reduced to logic, and provides a practical reason for being cautious in the use of logic for problem solving.

4. Soundness, Completeness and Heuristic Reasoning

Soundness assures the semantic accuracy of inference, requiring all provable assertions to be true, while completeness guarantees inference power, requiring all true assertions to be provable. However, soundness strongly constrains the granularity of thinking, while completeness restricts its semantic scope.

Sound reasoning cannot yield new knowledge; it can only make implicit knowledge explicit. Uncovering implicit knowledge may require creativity, for example when finding whether *P = NP* or Fermat's last theorem. But such creativity generally requires insights and constructions that go beyond deductive reasoning. The design and construction of software may likewise be viewed as uncovering implicit knowledge by creative processes that transcend deduction. The demonstration that a given solution is correct may be formally specified by "sound" reasoning, but the process of finding the solution is generally not deductive.

Human problem solvers generally make use of heuristics that sacrifice soundness to increase the effectiveness of problem solving. McCarthy suggested supplementing formal systems by a heuristic *advice taker* as early as 1960 [GR], but this idea has not yet been successfully implemented, presumably because the mechanisms of heuristic problem solving are too difficult to automate.

Heuristics that sacrifice soundness to gain inference power include inductive, abductive, and probabilistic forms of reasoning. Induction from a finite set of observations to a general law is central to empirical reasoning but is not deductively sound. Hume's demonstration that induction could not be justified by "pure reason" sent shock waves through nineteenth and twentieth century philosophy.

Abductive explanation of effects by their potential causes is another heuristic that sacrifices soundness to permit plausible

though uncertain conclusions. Choice of the most probable explanation from a set of potential explanations is yet another form of unsound heuristic inference. Inductive, abductive, and probabilistic reasoning have an empirical justification that sacrifices certainty in the interests of common sense.

Completeness limits thinking in a qualitatively different manner from soundness. Completeness constrains reasoning by commitment to a predefined (closed) domain of discourse. The requirement that all true assertions be provable requires a closed notion of truth that was shown by Godel to be inadequate for handling naturally occurring open mathematical domains like that of arithmetic. In guaranteeing the semantic adequacy of a set of axioms and rules of inference, completeness limits their semantic expressiveness, making difficult any extension to capture a richer semantics or refinement to capture more detailed semantic properties. Logic programs cannot easily be extended to handle nonformalized, and possibly nonformalizable, knowledge outside specific formalized domains.

The notion of completeness for theories differs from that for logic; a theory is complete if it is sufficiently strong to determine the truth or falsity of all its primitive assertions. That is, if every ground atom of the theory is either true or false. Theories about observable domains are generally inductive or abductive generalizations from incomplete data that may be logically completed by uncertain assumptions about the truth or falsity of unobserved and as yet unproved ground atoms (facts) in the domain. For example, the *closed-world assumption* [GN] assumes that every fact not provable from the axioms is false. Such premature commitment to the falsity of nonprovable ground assertions may have to be revoked when new facts become known, thereby making reasoning based on the closed-world assumption nonmonotonic.

Nonmonotonic reasoning is a fundamental extension that transforms logic into a more powerful reasoning mechanism. But there is a sense in which nonmonotonic reasoning violates the foundations of logic and may therefore be viewed as nonlogical. The benefits of extending logic to nonmonotonic reasoning must be weighed against the alternative of completely abandoning formal reasoning and adopting more empirical principles of problem solving, like those of object-oriented programming. Attempts to generalize logic to nonmonotonic or heuristic reasoning, while intellectually interesting, may be pragmatically inappropriate as a means of increasing the power of human or computer problem solving. Such extensions to deductive logic are better realized in a general computing environment than as extensions to logic programming languages.

Both complete logics and complete theories require an early commitment to a closed domain of discourse. While the closed-world assumption yields a different form of closedness than that of logical completeness or closed application programs, there is a sense in which these forms of being closed are related. In the next section the term *open system* is examined to characterize this notion as precisely as possible.

5. Open Systems

A system is said to be an *open system* if its behavior can easily be modified and enhanced, either by interaction of the system with the environment or by programmer modification.

1. A *reactive (interactive)* system that can accept input from its environment to modify its behavior is an open system.
2. An *extensible* system whose functionality and/or number of components can be easily extended is an open system.

Our definition includes systems that are reactive or extensible or both, reflecting the fact that a system can be open in many different ways. Extensibility can be *intrinsic* by interactive system evolution or *extrinsic* by programmer modification. Intrinsic extensibility accords better with biological evolution and with human learning and development, but extrinsic extensibility is the more practical approach to software evolution. The following characterization of openness explicitly focuses on this distinction:

1. A system that can extend itself by interaction with its environment is an open system.
2. A system that can be extended by programmer modification (usually because of its modularity) is an open system.

Since extrinsic extensibility is extremely important from the point of view of cost-effective life-cycle management, it is viewed as sufficient to qualify a system as being open. While either one of these properties is sufficient to qualify a system as being open, the most flexible open systems are open in both these senses.

Object-oriented systems are open systems in both the first and second senses. Objects are reactive server modules that accept messages from their environment and return a result. Systems of objects can be statically extended by modifying the behavior of already defined objects or by introducing new objects. Classes facilitate the abstract definition of behavior shared among a collection of objects, while inheritance allows new behavior to be defined incrementally in terms of how it modifies already defined behavior. Classes have the *open/closed* property [Me]; they are open when used by subclasses for behavior extension by inheritance, but are closed when used by objects to execute messages. The idea of *open/closed* subsystems that are both open for clients wishing to extend them and closed for clients wishing to execute them needs to be further explored.

Logic languages exhibiting don't-know nondeterminism are not open in the first sense, while soundness and completeness restrict extensibility in the second sense. To realize reactive openness concurrent logic languages abandon don't-know nondeterminism in favor of *don't-care nondeterminism*, sacrificing logical completeness.

Prolog programs can easily be extended by adding clauses and facts so they may be viewed as open in the second sense. But logical extension is very different from object-based extensibility by modifying and adding objects and classes. Because object-based languages directly model their domain of discourse, object-based extensibility generally reflects incremental extensions that arise in practice more directly than logical extension.

6. Don't-Care Nondeterminism

Don't-care nondeterminism is explicitly used in concurrent languages to provide selective flexibility at entry points to modules. It is also a key implicit control mechanism for realizing selective flexibility in sequential object-based languages. Access to an object with operations *op1, op2, ... , opN* is controlled by an implicit nondetermnistic select statement of the form:

select (op1, op2, ... , opN) endselect

Execution in a sequential object-based system is deterministic from the viewpoint of the system as a whole, but is non-

deterministic from the viewpoint of each object considered as an isolated system. The object does not know which operation will be executed next, and must be prepared to select the next executable operation on the basis of pattern matching with an incoming message. Since no backtracking can occur, the nondeterminism is don't care (committed choice) nondeterminism.

Concurrent porgramming languages like CSP and Ada have explicit don't care nondeterminism realized by guarded commands with guards Gi whose truth causes the associated body Bi to become a candidate for nondeterministic execution:

select (G1||B1, G2||B2, ... , GN||BN) endselect

The keyword *select* is used in place of the keyword *choice* to denote selective don't care nondeterminism, while guards are separated from bodies by || in place of |.

Guarded commands, originally developed by Dijkstra, govern the selection of alternative operations at entry points of concurrently executable tasks. For example, concurrent access to a buffer with an APPEND operation executable when the buffer is not full and a REMOVE operation executable when the buffer is not empty can be specified as follows:

select (notfull||APPEND, notempty||REMOVE) endselect

Monitors support unguarded don't-care nondeterminism at the module interface. Selection between APPEND and REMOVE operations of a buffer implemented by a monitor has the following implicit select statement:

select (APPEND, REMOVE) endselect

The monitor operations *wait* and *signal* on internal monitor queues notfull and notempty play the role of guards. Monitors decouple guard conditions from nondeterministic choice, gaining extra flexibility by associating guards with access to resources rather than with module entry.

Consider a concurrent logic program with a predicate P appearing in the head of N clauses of the form *"P(Ai) → Gi||Bi"*. A goal P(A) triggers nondeterministic execution of those bodies Bi for which A unifies with Ai and the guards Gi are satisfied. This execution rule can be specified by a select statement of the form:

select ((A1;G1)||B1, (A2;G2)||B2, ... , (AN;GN)||BN) endselect
Bi is a candidate for execution if A unifies with Ai and Gi is satisfied

Since no backtracking can occur once execution has committed to a particular select alternative, the nodeterminism is don't-care nondeterminism. However, don't care nondeterminism in concurrent logic languages is less flexible than in object-based languages because data abstraction and object-based message communication is not supported.

Don't-care nondeterminism is useful in realizing reactive flexibility, but is neither necessary nor sufficient for concurrent systems. Concurrent nonreactive systems for very fast computations are commonplace, while sequential object-based systems are reactive but not nonconcurrent. Reactiveness and concurrency are orthogonal properties of computing systems. Don't-care nondeterminism is primarily concerned with enhancing reactive flexibility and is not strictly necessary for concurrency.

Nondeterministic selection is relatively complex because it combines merging of incoming messages from multiple sources with selection among alternative next actions by pattern matching. The essential nondeterminism in concurrent systems arises from uncertainty about the arrival order (or processing order) of incoming messages and is modeled by implicit nondeterministic merging of streams rather than by explicit selection. For example, the nondeterministic behavior of a bank account with $100.00 when two clients each attempt to withdraw $75.00 depends not on selective don't-care nondeterminism but simply on the arrival order of messages from clients.

7. Are Concurrent Logic Programs Nonlogical?

Don't-care nondeterminism serves to realize reactive computations and also to keep the number of nondeterministic alternatives explored to a manageable size. But it may cause premature commitment to an inference path not containing a solution at the expense of paths that possibly contain solutions. Don't-care nondeterminism is nonmonotonic since adding a rule may have the effect of preventing commitment to an already existing rule. Logic programs employing don't-care nondeterminism are *incomplete* in the sense that they may fail to prove *true* assertions that would have been derivable by don't-know nondeterminism from the same set of clauses. It becomes the responsibility of the programmer to make sure that programs do not yield different results for different orders of don't-care commitment.

Under don't-care nondeterminism the result of a computation from a set of clauses depends on the order of don't-care commitment. This weakens the claim that concurrent logic languages are logical, reducing them to the status of ordinary programming languages. Clauses lose the status of inference rules, becoming mere computation rules. As hinted at in [Co], don't care nondeterminism takes the L out of LP, reducing logic programming to programming. The committed-choice inference paradigm loses the status of a proof technique and becomes a computational heuristic whose rules impose a rigid structure on both conceptualization and computation.

Don't-know nondeterminism provides a computational model for logical inference, while don't-care nondeterminism models incremental, reactive computation, but sacrifices logical inference. Reactive systems are open systems in the sense that they may react to stimuli from the environment by returning results and changing their internal state. Objects are a prime example of reactive systems, responding interactively to messages they receive. The inability of don't-know nondeterminism to handle reactiveness is a serious weakness of both logic programs and deductive reasoning. The fundamental reason for this is the inability of inference systems to commit themselves to incremental output.

While pure logic programming is incompatible with reactiveness it is definitely compatible with concurrency. The components of logical expressions may be concurrently evaluated. Universal and existential quantification, which is simply transfinite conjunction and disjunction, can be approximated by concurrent evaluation of components. Reactiveness is orthogonal to concurrency in the sense that concurrent nonreactive systems for very fast computations are commonplace, while sequential object-based systems are reactive but not nonconcurrent. However, reactive responsiveness is as important in large applications as concurrency. The identification of reactiveness and concurrency as independent goals of system design marks a step forward in our understanding of system requirements.

The *process interpretation* of concurrent logic programs views goal atoms as processes and logic variables as streams.

The set of goals at any given point in the computation becomes a dynamic network of processes that may be reconfigured during every goal-reduction step. Every concurrent logic program has a process interpretation, but concurrent object-oriented application programs cannot be directly mapped into concurrent logic programs. Thus concurrent logic programs are less expressive than object-oriented programs in the sense of [Sh1]. Logical processes have no local state; they are atomic predicates whose granularity cannot be adapted to the granularity of objects in the application domain. Concurrent logic programs give up their claim to be logical without gaining the communication and computation flexibility of traditional concurrent languages.

8. Are Multiparadigm Logic/Object Systems Possible?

Can the object-based and logic programming paradigms be combined to capture both the decomposition and abstraction power of objects and the reasoning power of logic? Experience suggests that logic is not by itself a sufficient mechanism for problem solving and that combining logical and nonlogical paradigms of problem solving is far harder than one might expect. Logic plays a greater role in verifying the correctness of programs than in their development and evolution. Finding a solution to a problem is less tractable than verifying the correctness or adequacy of an already given solution. For example, solutions of problems in NP can be verified in polynomial time but appear to require exponential time to find. Verification and validation is generally performed separately after a program (or physical engineering structure) has been constructed.

The logic and object paradigms have different conceptual and computational models. Logic programs have a clausal inference structure for reasoning about facts in a database, while object-based programs compute by message passing among heterogeneous, loosely-coupled software components. Logical reasoning is top-down (from goals to subgoals), while object-based design is bottom-up (from objects of the domain). Object-based programs lend themselves to development and evolution by incremental program changes that directly correspond to incremental changes of the modeled world. Inference rules provide less scope for incremental descriptive evolution, since rules for reasoning are not as amenable to change as object descriptions.

ICOT's choice of logic programming as the vehicle for future computing contrasts with the US Department of Defense's choice of Ada. Because Ada was designed in the 1970s, when the technology of concurrent and distributed software components was still in a primitive state, it has design flaws in its module architecture. But its goals are squarely in the object-oriented tradition of model building based on abstraction.

During the past 15 years we have accumulated much experience in designing object-oriented, distributed, and knowledge-based systems. The international computing community may well be ready for a major attempt to synthesize this experience in developing a standard architecture for distributed, intelligent problem solving in the 21st century. Such an architecture would be closer to the object-oriented than to the logic programming tradition.

Next-generation computing architectures should try to synthesize the logic and object-oriented traditions, creating a multiparadigm environment to support the cooperative use of both abstraction and inference paradigms. For example, an object's operations could in principle be implemented as logic programs, though the use of Prolog as an implementation language for object interfaces presents some technological problems. Perhaps technological progress in the 21st century will resolve these problems so that multiparadigm environments can be developed facilitating the cooperative application of both abstraction and inference paradigms.

Problem solving is a social process that involves cooperation among people, especially for large projects with a long life cycle. Decomposition of a problem into object abstractions is important both for cooperative software development and for incremental maintenance and enhancement. While object-oriented problem representation is not uniformly optimal for all problems, it does provide a robust framework for cooperative incremental software evolution for a much larger class of problems than logical representation.

The early optimism that artificial intelligence could be realized by a *general problem solver* gave way in the 1960s to an appreciation of the importance of domain-dependent knowledge representation. The debate concerning declarative versus procedural knowledge representation was resolved in the 1970s in favor of predicate calculus declarative representation. AI textbooks of the 1980s [CM, GN] advocate the predicate calculus as a universal framework for knowledge representation, with domain-dependent behavior modeled by nonlogical predicate symbols satisfying nonlogical axioms.

The logic and network approaches to AI have competed for research funds since the 1950s [Gr], with the logic-based symbol system hypothesis dominating in the 1960s and 1970s and distributed pattern matching and connectionist learning networks staging a comeback in the late 1980s [RM]. The idea that intelligence evolves through learning is an appealing alternative to the view that intelligence is determined by logic, but attempts to realize nontrivial intelligence by learning have proved combinatorially intractable. Distributed artificial intelligence research [BG] and Minsky's *The Society of Mind* [Min], view problem solving as a cooperative activity among distributed agents very much in the spirit of object-oriented programming. Ascribing mental qualities like beliefs, intentions, and consciousness to agents is likewise compatible with the object-oriented approach.

9. References

[BG] A. H. Bond and L. Gasser, *Readings in Distributed Artificial Intelligence*, Morgan-Kaufman 1988.

[Co] J. Cohen, Introductory remarks for the special CACM issue on logic programming, CACM, March 1992

[CM] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley 1984

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman 1978.

[GN] M. R. Genesereth and N. J. Nilson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufmann 1987.

[Gr] *The Artificial Intelligence Debate*, Editor Stephen Graubard, MIT Press 1988

[Me] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall International 1988.

[Min] Marvin Minsky, *The Society of Mind*, Simon and Schuster, 1987

[RM] D. E. Rumelhart and J. L. McLelland, *Parallel Distributed Processes*, MIT Press 1986.

[Sh1] E. Shapiro, Separating Concurrent Languages with Categories of Language Embeddings, TR CS91-05, Weizmann Institute, March 1991

[Sh2] E. Shapiro, The Family of Concurrent Programming Languages, *Computing Surveys*, September 1989

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Concurrent Logic Programming as a Basis for Large-scale Knowledge Information Processing

Koichi Furukawa

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
furukawa@icot.or.jp

1 The Future of Information Processing

As the Fifth Generation Computer Systems project claims, the information processing field is pursuing knowledge information processing.

Since the amount of information being produced is increasing rapidly, there is a growing need to extract useful information from this information. The most important and promising technologies for information extraction are knowledge acquisition and machine learning. They include such activities as classification of information, rule acquisition from law data and summary generation from documents. For such activities, heavy symbolic computation and parallel symbolic processing are essential.

Combinatorial problems are another source of applications requiring heavy symbolic computation. Human genome analysis and the inversion problems are examples of these problems. For example, in diagnosis, it is quite easy to forecast the symptoms given the disease. However, to identify the disease from the given symptoms is usually not so easy. We need to guess the disease from the symptom and to verify the truth of that guess by further observation of the system. If the system is linear, then the inversion problem is simply to compute the matrix inverse. But, in general, there is no straightforward way to solve the inversion problem. There may be many candidates for any guess and this becomes even worse when we take multiple faults into account. Note that abductive reasoning, one of the most important reasoning processes for open-ended problems, is also characterized as a general inversion formalism against deduction.

Cooperative problem solving (or distributed AI) is another important direction for future information processing. Like human society, one feasible way of dealing with large-scale problems is for a number of experts to cooperate. To exchange ideas between experts, mutual understanding is essential, for which we need complicated hypothetical reasoning to fill the gaps in terminology between them.

These three examples show the need for heavy concurrent information processing in the field of knowledge information processing in the future.

2 The Role of Logic Programming

Logic programming provides a basic tool for representing and solving many non-trivial artificial intelligence problems.

1. As a knowledge representation tool, it can express situations without being limited to a closed world, as was believed until recently. The negation by failure rule makes it possible to express an open-ended world, which is essential for representing common sense and dealing with non-monotonic reasoning. Recently, a model theory for dealing with general logic programs which contains negation-by-failure literals in the body of clauses has been studied. The theory, called stable model semantics, associates a set of feasible models, natural extensions of least models, to each general logic program.

2. As an inference engine, logic programming provides a natural mechanism for computing search problems by automatic backtracking or by an OR-parallel search mechanism. Recent research results show the possibility of combining top-down and bottom-up strategies for searching.

3. As a syntactic tool for non-deductive inference, logic programming provides a formal and elegant formalism. Abduction, induction and analogy can be naturally formalized in terms of logic and logic programming. Inoue et al. [Inoue *et al.* 92] showed that abductive reasoning problems can be compiled into proof problems of first order logic. This means that non-deductive inference problems can be translated into deductive inference problems. Since abduction is a formalization of a kind of inversion problems,

this provides a straightforward way to solve such problems.

There was a common belief that logic and logic programming had severe restrictions as tools for complex AI problems that require open endedness. However, recent research results shows they are expressive enough to represent and solve such problems.

3 The Role of Concurrent Logic Programming

Concurrent logic programming is a derivative of logic programming and is good for expressing concurrency and executing in parallel. From a computational viewpoint, concurrent logic programming only supports AND-parallelism, which is essential for describing concurrent and cooperative activities.

The reason why we adopted concurrent logic programming as our kernel language in the FGCS project is that we wanted simplicity in the design for our *machine language* for parallel processors. Since concurrent logic programming languages support only AND-parallelism, they are simpler than those languages which support both AND- and OR-parallelism.

We succeeded in writing many useful and complex application programs in KL1, the extension of our concurrent logic programming language, FGHC, for practical parallel programming. These include a logic simulator and a router for VLSI-CAD, and a sequence alignment program in genome analysis. These experimental studies show the potential of our language and its parallel execution technology.

The missing computational scheme in concurrent logic programming is OR-parallelism. It comes from the very fundamental nature of concurrent logic programming language, that is, the committed choice mechanism. OR-parallelism plays an essential role in many AI problems because of the requirement for searching. A great deal of effort has been made to achieve OR-parallel searching in concurrent logic programming by devising programming techniques. We developed three methods for different applications: a continuation-based method for algorithmic problems, a layered stream method for parallel parsing, and a query compilation method for database problems. These three methods cover many realistic applications. Therefore, we have almost developed OR-parallelism successfully. This means that there is a possibility of building parallel deductive databases in concurrent logic programming.

One of the most significant achievements using the query compilation method is a bottom-up theorem prover, MGTP [FujitaHasegawa 91]. This is based on the SATCHMO prover by [Manthey 88]. MGTP is a very efficient theorem prover which utilizes the full power of KL1 in a natural way by performing only one way unification. SATCHMO has a restriction in the problems it can efficiently solve: range-restrictedness [Furukawa 92]. However, most real life problems satisfy this condition and, therefore, it is very practical.

We succeeded in computing abduction, which was translated to a theorem proving problem in first order logic, by using MGTP. We succeeded in solving a very important class of inversion problems in parallel on our parallel inference machine, PIM.

4 Conclusion

Concurrrent logic programming gained its expressive power for concurrency at the sacrifice of Prolog's search capability. By devising programming techniques we have finally almost recovered the lost search capability. This means that we now have a very expressive parallel programming language for a wide range of applications.

As an example, we have shown that the technique enabled realization of an efficient parallel theorem prover, MGTP. We have also shown success in deductively solving an important class of inversion problems, formulated by abduction, by the theorem prover.

Our research results indicate that our concurrent logic programming and parallel processing based technologies have great potential for solving many complex future AI problems.

References

[FujitaHasegawa 91] H. Fujita and R. Hasegawa, *A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm.* In Proc. of the Eighth International Conference on Logic Programming, Paris, 1991.

[Furukawa 92] K. Furukawa, Summary of Basic Research Activities of the FGCS Project. In *Proc. of FGCS92*, Tokyo, 1992.

[Inoue *et al.* 92] K. Inoue, M. Koshimura and R. Hasegawa, Embedding Negation as Failure into a Model Generation Theorem Prover. To appear in *CAD-11: The Eleventh International Conference on Automated Deduction,* Saratoga Springs, NY, June 1992.

[Manthey 88] R. Manthey and F. Bry, *SATCHMO: A Theorem Prover Implemented in Prolog.* In Proc. of CADE-88, Argonne, Illinois, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Knowledge Information Processing in the 21st Century

Shunichi Uchida

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
uchida@icot.or.jp

1 A New Research Platform

Here in the last decade of the 20th century, the beginning of the 21st century is close enough for us to be able to forecast the kind of changes that will happen to new computer technologies and in the market and to predict what kind of research fields will be the most important.

I would like to try to forecast what will happen to parallel processing and knowledge information processing (KIP) based on my experience in the FGCS project.

It is quite certain that the following two events will happen;

1. Large-scale parallel hardware will be used for large-scale problem solving.
2. Symbolic processing and knowledge information processing applications will be extended greatly.

However, it is not so obvious whether these two events will be effectively combined or will remain separate. The key technology is new software technology to enable us to efficiently produce large and complex parallel programs, especially for symbolic and knowledge processing applications. If this parallel software technology is provided with large-scale parallel hardware, a very large change will happen in the market in the 21st century. I think that the FGCS project has developed the kernel of this key technology and shown that these two events will surely be combined.

In the FGCS project, we proposed the hypothesis that a logic programming language family would be superior to any other language families in exploiting new software technology and applications, especially for symbolic and knowledge information processing. The first step in proving this hypothesis was to show that the above two events can be smoothly combined by logic programming. We decided to design and implement a logic language on large-scale parallel hardware.

In designing and implementing this logic language, the most important problem was to find an efficient method to realize the following two very complex mechanisms;

1. An automatic process synchronization mechanism based on a dataflow model
2. An automatic memory management mechanism including an efficient garbage collection method for distributed memories

These mechanisms greatly reduce the burden of parallel programming and are indispensable for implementing not only a parallel logic language but also any other high-level language including functional language such as a parallel version of LISP.

We have developed a parallel logic language, KL1, its language processor and programming environment, and a parallel operating systems, PIMOS. These are now implemented on parallel inference machine hardware, PIM hardware, which connects up to 512 processing elements. We have also developed a parallel DBMS called Kappa-P on the PIMOS. We call all of these software systems FGCS basic software.

Through the development of experimental parallel application systems using this basic software, we have already experienced that we can efficiently produce parallel programs which make full use of the power of parallel hardware.

This basic software is now available only on PIM hardware which has some hardware support to make KL1 programs run faster such as tag handling support or a large capacity main memory. However, recently, it has been announced that many interesting parallel hardware systems are to appear in the market as high-end supercomputers aiming at large-scale scientific calculations. Some of them have an MIMD architecture and employ a RISC type general purpose microprocessor as their processing element.

It is certain that the performance and memory capacity of these processing elements will increase in the next few years. At that stage, it will be possible to implement the FGCS basic software on this MIMD parallel hardware and obtain reasonable performance for symbolic and knowledge processing applications. If this is implemented, this parallel hardware will have a high-level parallel logic programming environment combined with a conventional programming environment.

This new environment should provide us with a powerful and widely-usable common platform to exploit knowledge information processing technology.

2 KIP R&D in the 21st Century

2.1 Knowledge representation and knowledge base management

The first step to proving the hypothesis that the logic language family is the most suitable for knowledge information processing is to obtain a new platform for further research into knowledge information processing. For this step, a low-level logic language, namely, KL1 was developed.

The second step is to show that a logic language will exploit new software technology to handle databases and natural knowledge bases. The key technology in this step will be knowledge representation and knowledge base management technology.

Using a logic language as the basis for knowledge representation, it should be a natural consequence that the knowledge representation language has the capability of performing logical deduction.

Users of the language will consider this capability desirable for describing knowledge fragments, such as various rules in our social systems and constraints in various machine design. The users may also want the language to have been an object-oriented modeling capability and a relational database capability, as built-in functions.

Currently, we do not have good criteria to combine and harmonize these important concepts and models to realize a language having these rich functions for knowledge representation.

The richness of these language capabilities will always impose a heavy overhead on its language processor. The language processor in this case is a higher-level inference engine built over a database management system. It is interesting to see how much the processing power of parallel hardware will compensate for this overhead.

In the FGCS project, we developed a database management system, Kappa-II based on the nested relational model. It was implemented on a sequential inference machine, PSI, for the first time. Now, its parallel version, Kappa-P written in KL1, has been built on the PIM hardware. Over Kappa-P, we have designed a knowledge representation language, Quixote and a KBMS based on the deductive and object-oriented model. Its first implementation has been completed and is now under evaluation. Quixote is one of the high-level logic languages developed over KL1. These evaluation results should provide very interesting data for forecasting database research at the beginning of the 21st century.

Another high-level logic language developed in the FGCS project is a parallel constraint logic programming language, GDCC. GDCC has a constraint solver in its language processor which can be regarded as an inference engine dedicated to algebraic problem solving.

Another kind of inference engine is a parallel theorem prover for first order logic which is called a model generation theorem prover, MGTP. This prover is now used as the kernel of a rule-based reasoner in a law expert system, also known as the legal reasoning system, Hellic-II.

These logic languages and inference engines will be further developed during this decade. They will be implemented on large-scale parallel hardware and will be used as important components to organize a new platform to build a knowledge programming environment in the first decade of the 21st century.

2.2 Knowledge programming and knowledge acquisition

The third step to proving the hypothesis is to show that a knowledge programming environment based on logic programming will efficiently work to build knowledge bases, namely, the contents of a KBMS.

Knowledge programming is a programming effort to translate knowledge fragments into internal knowledge descriptions that are kept and used in a KBMS.

This process may be regarded as a conversion or compiling process from "natural" knowledge descriptions, which exist in our society for us to work with, into "artificial" knowledge descriptions, which can be kept in the KBMS and used efficiently by application systems such as expert systems. If this process is done almost automatically by some software with a powerful inference engine and knowledge base, it is called "knowledge acquisition". Some people may call it "learning".

In human society, we have many "natural" knowledge bases such as legal rules and cases, medical care records, design rules and constraints, equipment manuals, language dictionaries, various business documents and rules and strategies for game playing. They are too abstract and too context-dependent for us to translate them into "artificial" knowledge descriptions.

In the FGCS project, we developed several experimental expert systems such as a natural language processing system, a legal reasoning system, and a Go playing system. We have learned much about the problems of how to code or program a "natural" knowledge base, how to structure knowledge fragments to be able to use them in application programs, and so on.

We have also learned that there is a big gap between the level of "natural" knowledge descriptions and that of the "artificial" knowledge descriptions which current software technology can handle. We were forced to realize again that "natural" knowledge bases have been built not for computers but for human beings. The existence of this large gap means that current computer technology is not intelligent enough to accept such knowledge bases.

It is obvious that more research effort is needed to build much more powerful inference engines that will provide us with much higher-level logical reasoning functions based on formal and informal models such as CBR,

ATMS and inductive inference. In parallel with this effort, we have to find some new methods of preprocessing "natural" knowledge descriptions to obtain more well-ordered forms and structures for "artificial" knowledge bases. For example, we have to create new theories or smodeling techniques to explicitly define context-dependent information hidden behind "natural" knowledge descriptions. The situation theory will be one of these theories.

It is interesting to see how these powerful inference engines will relate to knowledge representation language and knowledge structuring methods. Another interesting question will be to what extent the power of larger-scale parallel hardware and parallel software technology will make these higher-level inference functions practical for real applications.

It is certain that research into knowledge information processing will continue to advance in the 21st century, opening many new research fields as it advances and leaving a large growing market behind it.

ICOT SESSIONS

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

LSI-CAD Programs on Parallel Inference Machine

Hiroshi Date† Yukinori Matsumoto† Koichi Kimura†
Kazuo Taki† Hiroo Kato‡ Masahiro Hoshi‡

†Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan
{date, yumatumo, kokimura, taki}@icot.or.jp

‡Japan Information Processing Development Center
3-5-8, Shibakouen, Minato-ku, Tokyo 105, Japan
{j-kato, hoshi}@icot21.icot.or.jp

Abstract

This paper presents three kinds of parallel LSI-CAD systems developed in ICOT and describes their experimental results on a parallel inference machine. These systems are routing, placement and logic simulation. All of them are implemented in KL1, a concurrent logic language, and executed on the Multi-PSI, a distributed memory machine with 64 processors.

We regard our parallel inference machines as high performance general purpose machines. We show programming techniques to derive high performance on parallel inference machines. The common objectives of these systems are, firstly, to provide speedup by extracting major parallelism, and, secondly, to show the applicability of our hardware and language system to practical applications. For this reason, our systems are evaluated using real LSI chip data.

The key features are, in the routing system, *concurrent object modeling* of routing problems to realize a lot of concurrency; in the placement system, *time-homogeneous parallel simulated annealing* to optimize placement results; and in the logic simulation system, *the Time Warp mechanism* as a time-keeping mechanism for simulations.

Experimental results of these systems show that these techniques are effective for parallel execution on large-scale MIMD machines with distributed memory structure, like the parallel inference machines.

1 Introduction

A parallel computer system PIM (the Parallel Inference Machine), one of the goals of the Japanese Fifth Generation Computer Systems project, has been completed, and its evaluation is starting. PIM has been developed mainly to target high performance knowledge information processing. Since most problems in this domain are of an extremely large size, exploiting the whole power of parallel machines is important. In practice, however, it is not easy to derive their maximum power because of the non-uniformity of computation, that is dynamically changing parallel computation depending on time and space.

In order to move programs efficiently on PIM, the following are important. First is to adopt good concurrent algorithms. Second is to design programs based on programming paradigms to realize high parallelism. And last is to use effective load distribution techniques including processor mapping. We aimed at gaining experiences with these techniques through large-scale practical application experiments on PIM.

PIM is an *inference* machine, however, its applicability should not be limited to knowledge information processing. From the viewpoint that PIM is a high performance general purpose machine, we chose LSI-CAD as one of the application fields.

Nowadays, LSI-CAD is indispensable for LSI design. The integration of the LSI chip has increased exponentially in proportion to the progress of the semiconductor process technology. The quality of LSIs depend on the performance of LSI-CAD tools. Therefore, higher performance is required. Besides, the flexibility of the tools must be kept for a variety of demands. Using hardware accelerators is one possible way of obtaining faster tools, however, it usually results in a sacrifice of flexibility. A likely alternative is to parallelize software tools. This certainly satisfies the above two requirements: making the tools faster and keeping their flexibility.

We focused on three stages of LSI-CAD; *logic simulation, placement* and *routing*, which are currently the most time-consuming in LSI design. Each system has following features.

The routing system finds paths based on the lookahead line search algorithm [Kitazawa 1985]. This algorithm provides high quality solutions, however, it was originally proposed with the assumption of sequential execution. We introduced a new implementation method of parallel

router based on the concurrent objects model, and improved the basic algorithm to make it suitable for parallel execution. The concurrent objects model is expected to derive a lot of parallelism among small granular processes. We investigated the description complexity and overhead of our routing programs. Also its performance (real speed, speedup and wiring rate) was evaluated in comparison with a sequential router on general purpose computer using real LSI chip data.

The cell placement problem is a combinatorial optimization problem. Simulated annealing (SA) is a powerful algorithm to solve such problems. Cooling schedules are important for efficient execution of SA. In our placement system, the time-homogeneous parallel SA algorithm [Kimura *et al.* 1991] was adopted. This algorithm constructs appropriate cooling schedules automatically. We evaluated quality of solutions in our system using MCNC benchmark data.

Logic simulation is an application of discrete event simulation. The key to its efficient execution in parallel is keeping the time correctness without large overheads. We adopted the Time Warp mechanism (TW) as the time-keeping mechanism. TW has been considered to contain large rollback overheads, however, it has not been evaluated in detail yet. We not only improved the rollback process but also added some devices so that TW would become an efficient time-keeping mechanism. Cascading-Oriented Partitioning strategy for partitioning circuits are also proposed to attain good load distribution. We evaluated our system on speedup and real speed (events/sec) as compared with the systems that had other time-keeping mechanisms (Conservative and Time Wheel) using ISCAS89 benchmark data.

These systems were implemented in KL1 [Chikayama *et al.* 1988, Ueda *et al.* 1990], a concurrent logic language, and have been experimented with on the Multi-PSI/V2 [Nakajima *et al.*1989, Taki 1988], a prototype of PIM.

This paper is organized as follows : The routing system is described in Section 2. A routing algorithm based on the concurrent objects model and its implementation is presented in detail. Section 3 explains the placement system. The time-homogeneous SA algorithm is introduced and optimization in the implementation is explained. Section 4 overviews the logic simulator and reports on its evaluation. Our conclusion is given in Section 5.

2 Routing System

2.1 Background

There have been many trials to realize high speed and good quality router systems with parallel processing. These trials can be classified into two areas. One is the hardware engine which executes the specified routing algorithm efficiently [Kawamura *et al.* 1990, Nair *et al.* 1982, Suzuki *et al.* 1986]. The other is concurrent routing programs implemented on general purpose parallel machines [Brouwer 1990, Olukotun *et al.* 1987, Rose 1988, Watanabe *et al.* 1987, Won *et al.* 1987]. The former approach can realize very high speeds, while the latter can provide large flexibility. We took the latter approach to realize both high speed and a flexible router system, targetting very large MIMD computers.

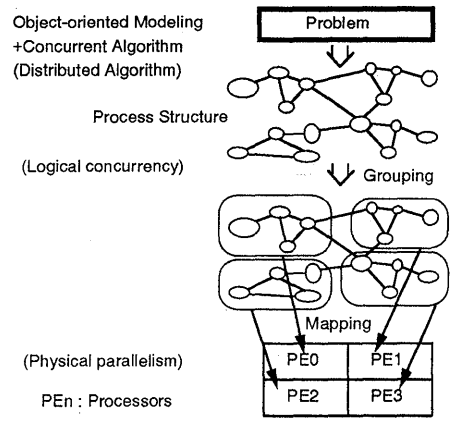


Figure 1: Program design paradigm based on the concurrent objects model

In general, a lot of parallelism is needed to feed a large MIMD computer. So, we propose a completely new parallel routing method, based on a small granual concurrent objects model. The routing method was implemented on the distributed memory machine, Multi-PSI, with a logic programming language KL1. We made preliminary evaluations of the new router, from the viewpoints of (1) data size vs. efficiency, (2) wiring rate vs. parallelism, and (3) comparison of execution speed with general purpose computers.

This section contains the following. A programming paradigm based on the concurrent objects model, a router program with an explanation of concurrent algorithms and implementation, problems in parallelization, and preliminary measurements and evaluation results.

2.2 Programming Paradigm

Formalizing a problem based on *the concurrent objects model* is one of the most prospective ways to embed parallelism in a given problem.

This section describes our methodology to design parallel programs from problem formalization to parallel execution. We also show coding samples in the KL1 language.

Figure 1 shows the flow of parallel program design. Firstly, a given problem is formalized based on the concurrent objects model. That is, many objects make a solution cooperatively, by exchanging messages. At the same time, a concurrent algorithm is designed upon the

```
Concurrent_Object ( [ Message_1 | Rest ], Interior state variables, Stream variable ) : -
    true |
        Process correspond to Message_1,
        Renew of Interior state variables,
        Stream variable = [Message | New Stream variable],
        Concurrent_Object ( Rest , New Interior state variables, New Stream variable ).

Concurrent_Object ( [ Message_2 | Rest ], Interior state variables, Stream variable ) : -
```

Output of Message to other objects

Figure 2: Implementation of a concurrent object in KL1

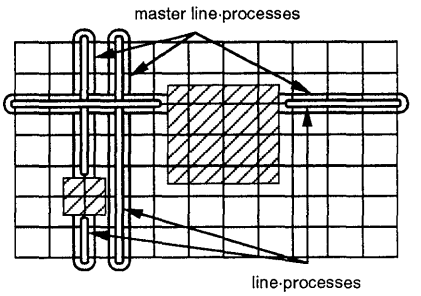


Figure 3: Master line processes and line processes

model. Sometimes, the algorithm is a distributed algorithm. Through this design phase, the activities of the objects corresponding to messages are defined.

Then, each object is implemented as a KL1 process. Process connection topology is decided based on input data. Usually, a much larger number of processes is needed than the number of processors to get good load balance. Logical concurrency (the possibility of parallel processing) is designed through this flow.

Secondly, the processes, which exchange messages frequently, are grouped to increase communication locally. When each process has a large computational amount (large granularity) and a low communication rate, this phase can be omitted.

Then, the groups are assigned to processors and executed. This is called mapping. Physical parallelism is realized in this phase. The KL1 language system allows independent descriptions of the problem solving part (logical concurrency) and the mapping part (physical parallelism) of a program. Performance tuning of parallel processing can be done only by changing the mapping part, not by changing the problem-solving algorithm.

The KL1 language is quite suitable for describing concurrent objects. Processes representing the objects are written in the self recursive call of the KL1 language. These processes can communicate with each other through the message streams. Figure 2 shows a coding sample of an object. The functions of an object are defined with a set of clauses. Each clause corresponds to a message which the object receives.

2.3 Router Program

We used the lookahead line search method [Kitazawa 1985] as a basic algorithm. Then we reconstructed the algorithm for highly parallel execution, taking the concurrent objects model as a basic design framework.

2.3.1 Basic algorithm

The lookahead line search method is one of the line search algorithms coupled with lookahead operation. It is, if you like, a sort of hill-climbing algorithm, looking for a good route. The algorithm, also, has two features. One is to escape from the local optimum point with the help of the Inhibited Expected Point (IEP) flag. The other is backtracking to retrace bad routes and to retry searching. The algorithm guarantees connection between a start point and a target point when paths exist between them.

2.3.2 Concurrent routing algorithm

In KL1 programming, an execution unit is a process corresponding to an object. Since the line search algorithm decides a route, line by line, we designed the concurrent algorithm so that objects=processes corresponds to every line segment on a routing grid. Line processes exchange messages with each other to look for a good route. Each line process maintains the corresponding line's status and, at the same time, the execution entity of the search.

As Figure 3 shows, each process corresponds to each grid line (*master line process*) and line segment (*line process*) on it. A master line process manages line processes on the same grid line and passes messages between the line processes and crossing line processes.

The routing procedure of one net is almost the same as that of the basic algorithm, except that the procedure is broken down into a sequence of messages and their operations are executed among processes. Computing the best expected point is done as follows. The expected point is the closest location to a goal on a line segment. The distance to the goal is used for a cost function in the hill climbing method.

When a line process receives a routing request message with information of a goal point, it changes its status to "under searching". Then, it sends request messages for calculation of expected points to line processes that cross it (Figure 4).

Thus computation of the expected point is executed concurrently on each line process that receives the request message.

After the computation results are returned to the searching line process, it aggregates those results and determines the best expected point.

When the best expected point is determined, the searching line is connected to the crossing line that includes the best expected point. The searching line process splits into an occupied part and a free part, and

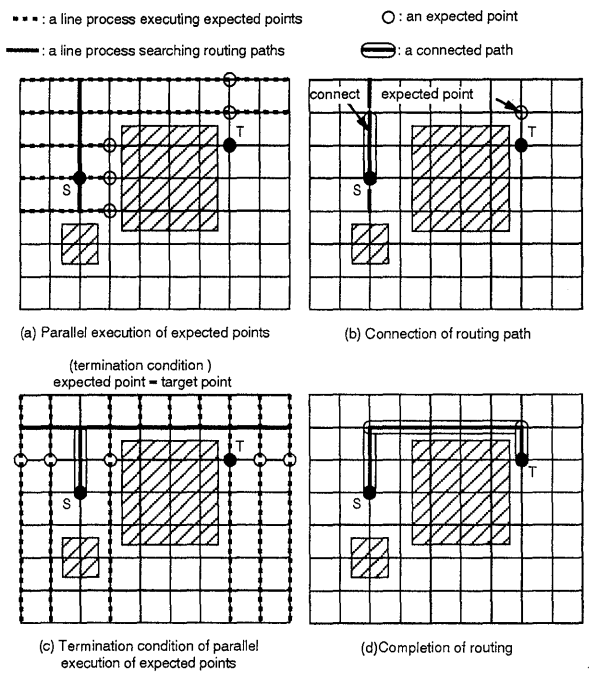


Figure 4: Parallel execution of expected points

the status is maintained. Then, the next routing request message is sent to the connected crossing line.

Messages are sequenced at the entrance of each process. Only one message can be handled at any time in a process. No problems of exclusive access to an object or locking/unlocking objects arise with this scheme.

In our algorithm, two types of parallelism are embedded. One is concurrent computation in the lookahead operation and the other is concurrent routing of different nets.

2.4 Problems in Parallel Execution

When we parallelize the lookahead line search method, three problems arise. The first is deadlock, the second is conflict among routing nets and the last is memory overflow for communicating between processors.

2.4.1 Avoidance of deadlock

When two or more nets are searched concurrently, deadlock may occur. Figure 5 shows an example. When line processes that intersect orthogonally send request messages to compute the expected point to each other at the same time, computation will not occur. This is because they cannot carry out the next messages until the execution of present messages terminates.

If it is guaranteed that execution of a message terminates within a fixed period, deadlock can be avoided. To

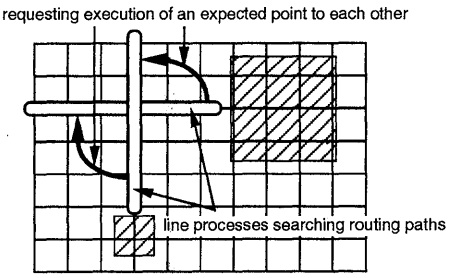


Figure 5: Example of deadlock

satisfy this condition, we made the following modification.

Firstly, messages are grouped into group A and group B. B-type messages are guaranteed to terminate execution within a fixed period. A-type messages are not guaranteed to terminate, that is, some synchronization with other processes is needed before terminating message execution.

We modify the operations of A-type messages as follows. Each process executing an A-type message observes all messages arriving successively. When an A-type message is found, it is left in a message queue, that is no operations are performed. When a B-type message is found, it is processed immediately before termination of the currently executing A-type message. For this processing of B-type messages, a temporary process status that differs from the sequential algorithm is needed. By applying this modification, deadlock can be avoided.

In our router, the routing request messages are A-type, and the request messages of computing expected points are B-type.

2.4.2 Conflict among nets

When concurrent routing of multiple nets is done, different nets may conflict on the same line segment. In this situation, message sequencing works well and the first message to arrive (corresponding to net A) occupies the segment. The second message to arrive (net B) fails to complete a route, and backtracks.

However, net A may backtrack afterwards and may release the line segment. In this case, net B does not visit the line segment anymore and the line segment may be left unused. This fact causes lower quality routes (longer paths) or a lower wiring rate (more unconnected nets).

To avoid those degradations in routing quality, the scheduling of the order in which nets start routing is important and may limit concurrency by eliminating the number of nets routed concurrently. However, parallelism can be affected by these controls. Relations between wiring rates and parallelism are studied in the experiments.

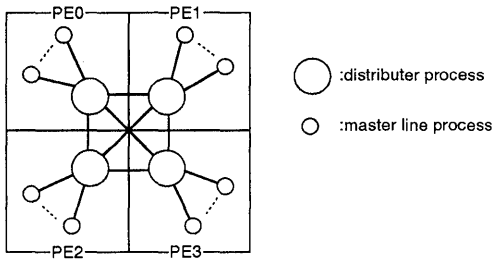


Figure 6: Improved process structure

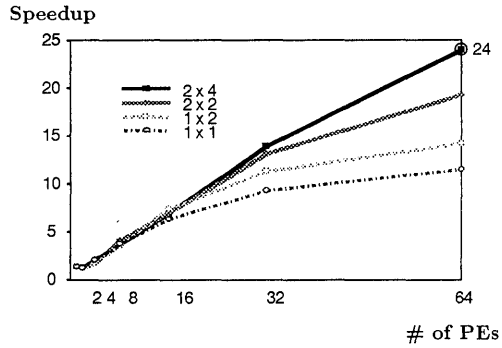


Figure 7: Data size vs. speedup

2.4.3 Overflow of memory for communication among processors

When we implement the concurrent program using KL1, two kinds of memory are necessary. One is the memory for representing processes. The other is the memory for communication paths among processors.

In our routing program, the process structure shown in Figure 3 was implemented. Each master line processes, which mediates between line processes, must communicate all orthogonal master line processes. Therefore the number of communication paths is increasing for large-scale data. Experimental results show that the maximum grid size of chip data to be treated by this routing program is about 500 × 500. This size is too small for applying practical data.

In order to solve this problem, we improved the process structure, as in Figure 6. Each distributer process controls communication among processors.

2.5 Measurements and Evaluation

We evaluate our router from the following three points of view. (1) Data size vs. Speedup, (2) Parallelism vs. Wiring Rate, and (3) Comparison with a general purpose computer. The program was executed on a MIMD machine with distributed memory and 64 processors, the Multi-PSI. Two types of real LSI data were used. The features of these data are shown in Table 1. Terminals to be connected are distributed uniformly in DATA1. Meanwhile, terminals are concentrated locally in DATA2. DATA3 is large-scale data.

Table 1: Testing data

Data	DATA1	DATA2	DATA3
Grid size	262×106	322×389	2746×3643
# of nets	136	71	556
Presented by	Hitachi Ltd.	NTT Co.	NTT Co.

2.5.1 Data size vs. speedup

Generally, when data size increases, the number of processes increase too and more parallelism can be expected. Higher parallelism can lead to greater efficiency or larger speedup with a fixed number of processors. We measured the relationship between the size of data and speedup.

In this experiment, we used data copying DATA1. Here we measured four cases (1×1, 1×2, 2×2, and 2×4). Figure 7 shows the result of measurement. This graph shows that the larger the size of data, the higher the speedup. It also shows 24-fold speedup with 64 processors for 2×4 data it does not look saturated yet. We have to investigate the limit of speedup with increasing data size.

2.5.2 Wiring rate vs. parallelism

Parallel routing of multiple nets may cause a degradation in wiring rate. We measured the relation between wiring rate and parallelism for DATA1 and DATA2, as shown in Figure 8. The two vertical axes show execution time and wiring rate. The horizontal axis shows the number of nets routed concurrently. Parallelism is proportional to this. When equal to one, parallelism only arises from parallel lookahead operations. It was observed that terminal-distributed data shows good wirability, even if parallelism is high, when the terminal-concentrated data is poor. Concentrated terminals tend to cause a lot more net confliction.

2.5.3 Comparison with a general purpose computer

The execution time of DATA2 with a single processor was measured as 111 seconds. From Figure 8, speedup caused only by lookahead operation is calculated as 4.9.

The execution time of our system was compared with a general purpose computer, the IBM 3090/400, which is a 15 MIPS machine. The sequential lookahead line search router on the IBM machine was developed by Dr. Kitazawa (NTT Co.) before our work was conducted. Table 2 shows the performance of the routers.

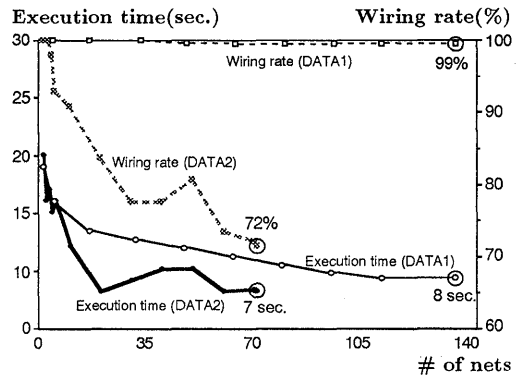


Figure 8: Wiring rate vs. parallelism

Two cases of the Multi-PSI measurements (with 64PEs) are included in the table. One routed all nets concurrently and the other routed each net one after another. The former case shows the better execution time but worse wiring rate. The latter case accomplished the perfect wiring rate but worse execution time for DATA2. We expect to realize both good execution time and good wiring rate by controlling the number of nets wired concurrently and changing the wiring order. (In fact, on DATA2, W:100 % and E:16 sec. under the number of nets wired concurrently is equal to 2.)

The evaluation for large data (DATA3) has just started. The wiring rate in the table is still insufficient but it will be improved as mentioned just above.

Table 2: Comparison of performance

Data\Machines		IBM 3090/400	Multi-PSI (64PEs) †	Multi-PSI (64PEs) ‡	Multi-PSI (1PE)
DATA2	E	7.45	7.0	20.0	111.0
	W	100	72	100	100
DATA3	E	405.0	360.0	N.A.	N.A.
	W	100	90	N.A.	N.A.

E:execution time (Sec.),W:wiring rate(%)
† concurrent wiring of all nets
‡ sequential wiring of each net

The execution time of the router on the Multi-PSI can be considered almost comparable with that on an IBM machine. When our router is ported to PIM machines, the next model to the Multi-PSI, the execution time will be reduced to 1/10 to 1/20 in execution with 256 to 512 processors.

The performance of the bare hardware of a Multi-PSI processor is 2 to 3 MIPS. And the efficiency of parallel processing (speedup/number of processors) is 25% for the case of Multi-PSI(64PEs) with concurrent wiring of all nets on DATA2. So, bare hardware performance with 64 processors is expected to be 32 to 48 MIPS (2 to 3 $\times$ 64 $\times$ 0.25). While, the actual performance is comparable with the 15MIPS machine. The degradation of actual performance must be caused by the implementation overhead of the object-oriented program and KL1 language.

2.6 Discussions

We presented a new routing method based on the concurrent objects model, which can include very large concurrency and is suitable for very large parallel computers. The program was implemented on a distributed memory machine with 64 processors. Preliminary evaluation was then done with actual LSI data.

The experimental results showed that the larger the data size, the higher the efficiency attained by a maximum of 24-fold speedup with 64 processors against single processor execution. The speedup curve did not look significantly saturated, that is, more speedup can be expected with more data.

In experiments on parallelism and the wiring rate, a good wiring rate with large parallelism was attained for data in which terminals are distributed uniformly. However, for data with concentrated terminals, the wiring rate became significantly worse, due to the increase in parallelism. We must improve the wiring rate in the latter case.

The actual performance of our router system was compared with an almost identical router on a high-end general purpose computer (IBM3090/400, 15 MIPS). Results showed that the speed of both systems was comparable. Based on a rough comparison of bare hardware speeds, the implementation overheads of the parallel object-oriented program and our language are estimated as 100 to 200% in total, against the sequential FORTRAN program on the IBM machine.

3 Placement System

3.1 Background

Cell placement is the initial stage of the LSI layout design process. After the functional and logical designs of the circuit are completed, the physical positions of the circuit components are determined so as to route all electrical connections between cells in a minimum area without violating any physical constraints. Heuristics for evaluating the quality of a placement usually promote one or more of the following: minimum estimated wire length, an even distribution of wires around the chip, minimum layout area, and regular layout shape.

The cell placement problem is well-known as a difficult combinatorial optimization problem. In other words, it is not feasible for obtaining the *optimum* placement of a circuit with practical size because it takes excessively amounts of CPU time. So efficient techniques to get nearly *optimum* placement must be employed in practice.

3.2 Simulated Annealing

Approximate methods are used to solve the combinatorial optimization problem. One such method is called *iterative improvement*. In this algorithm, the initial solution is generated, and, then, modified repeatedly to try to improve it. In each iteration, if the modified solution is better than the previous one, the modified solution becomes the new solution.

The process of altering the solution continues until we can make no more improvement, thus yielding the final solution. The problem with this algorithm is that it can be trapped at a *local optimum* in a solution space.

The Simulated Annealing(SA) algorithm [Kirkpatrick *et al.* 1983] is proposed to solve this problem. It probabilistically accepts a new solution even if the new solution may be worse temporarily. Its acceptance probability is calculated according to the change in the estimated cost value of the solution and the parameter "temperature". The cost function is often referred to as "energy". In this way, it is possible to search for the *global optimum* without being trapped by local optima.

The details of this algorithm are as follows.

It is constructed from two criteria, the *inner loop* criterion and the *stopping* criterion. At first, the initial solution and initial temperature are given. In the inner loop criterion, new solutions are generated iteratively and each solution is evaluated to decide whether it is acceptable. The units of iteration which are constructed by generating and estimating the new solution are called "step". In each stage of the inner loop criterion, the temperature parameter is fixed. In the stopping criterion, after a sufficient number of iterations are performed in the inner loop, the temperature is decreased gradually according to a given set of temperatures called the "cooling schedule". The stopping criteria are satisfied when the energy no longer changes.

One of the most difficult things in SA is finding an appropriate cooling schedule, which largely depends on the given problem. If the cooling schedules are not adequate, satisfiable solutions will never be obtained.

3.3 Parallel Simulated Annealing

A new parallel simulated algorithm(PSA) is proposed to solve the cooling schedule problem [Kimura *et al.* 1991]. The most important characteristic of this algorithm is that it constructs the cooling schedule automatically from the given set of the temperatures. The basic idea is to use parallelism in temperature, to perform SA processes concurrently at various temperatures instead of sequentially reducing the temperature. So it is scheduleless or *time-homogeneous* in the sense that there are no time-dependent control parameters.

After executing a fixed number of annealing steps, the solutions between the adjacent temperatures are probabilistically exchanged as follows. When the fixed number of annealing steps is denoted by k, $1/k$ is called the "frequency". When the energy of the solution at a higher temperature is lower than that at a lower temperature the solutions between these temperatures are exchanged unconditionally. Otherwise they are exchanged according to a probability that is determined by differences in their energies and temperatures [Kimura *et al.* 1991].

In PSA, even if a solution is trapped at a *local optimum* at a certain temperature, it is still possible to search for *global optima* because another new solution can be supplied from a higher temperature. So a *nearly optimum* solution will finally be found at the lowest temperature.

3.4 Outline of the System

Our experimental placement system employs the PSA algorithm. It is constructed on Multi-PSI, an MIMD machine, and the KL1 language is used to implement the system [Chikayama *et al.* 1988]. The intention is to provide a satisfiable solution in a feasible time. It is also applied to placement problems to examine the efficiency of the PSA algorithm.

The object of this system is the standard cell LSI without any macro blocks. The standard cells have uniform height and variant widths. These cells are arranged in multiple cell-blocks so as to minimize the chip area. Namely, it decides the location of each cell so as to minimize the total estimated wire length, which approximates the total routing length.

3.5 Implementation

3.5.1 Initial placement and new solution generation

The initial cell positions are determined randomly. In our placement system, SA processes are split into two temperature regions. The number of temperatures in the two regions should be specified by the user. Usually one or more temperatures are necessary for the lower region.

In the higher temperature region, there are two ways to generate a new solution. One way is to move a randomly selected cell to a random destination. The other way is to exchange the position between two randomly selected cells.

In the lower temperature region, generating a new solution is done by exchanging two arbitrary adjacent cells within a cell-block.

Moreover the range-limiter window is introduced [Sechen *et al.* 1985]. The range-limiter window restricts the ranges for moving and exchanging cells. The lower the temperature becomes, the smaller the size of the window becomes. It suppresses the generation of new solutions that are unlikely to be accepted.

3.5.2 Estimation of a new solution

The energy of a solution is the sum of the three values listed below [Sechen *et al.* 1985].

- estimated wire length
- the cell overlap penalty
- the block length penalty

The estimated wire length approximates the routing lengths between the cells. The estimated wire length of a single net is the half-perimeter length of the minimum bounding box which encloses all of the pins comprising the net.

The cell overlap penalty estimates the overlap between cells. In the higher temperature region, we permit overlap between cells because the cost of recalculating the estimated wire length for a new solution can be reduced. If overlap were not permitted, the overlap incurred by moving or exchanging cells would have to be removed by shifting many cells. As a result, the estimated wire length would have to be recalculated with respect to all of the nets connected to these shifted cells. In the lower temperature region, cells are never overlap, a new solution can be re-estimated only by calculating the change in total estimated wire length, because the two penalties don't change in this case.

The block length penalty estimates the difference between ideal and real block length. It is desirable to have cell blocks of a uniform length.

When the solutions are exchanged between the two temperature regions, the overlap between cells in the higher temperature region is removed as the solution is passed to the lower one.

3.5.3 Load distribution and solution exchange between adjacent temperatures

In PSA, each SA process is assigned to a separate processor, because executions at each temperature are highly independent and the amount of execution is nearly equal.

When we try to implement the exchange mechanism of the solutions, the natural way may be to exchange the solutions between the processors. But, when an MIMD machine like Multi-PSI is used, the exchange of large placement data between processors incurs a large communication overhead. So, solutions exchange between adjacent temperatures should be done by exchanging temperature values between processors.

Processors with adjacent temperatures hold a common variable and use this for communication. This is called a "stream" in KL1 [Chikayama *et al.* 1988] and is realized by an endless 'list'. These streams are also swapped between processors when the solutions between adjacent temperatures are exchanged.

3.5.4 Performance monitoring subsystem

The monitor displays the energy value of each SA process in real-time. It is useful to overview the entire state of the system performance. This energy graph is updated when adjacent temperatures are exchanged. As it displays the exchange in energy value in real time, it helps us to decide when to stop the execution. After several short time executions, we can decide the number of temperatures in the two regions and the highest temperature from the dispersion of the energy graph.

The monitoring subsystem is constructed on a Front End Process so that it does not incur an overhead in SA process execution. It is also possible to roll back the energy graph while SA processes are being executed.

Table 3: Number of temperatures .vs. quality of solution
inner-loop count = 20,000 times, frequency of exchange = 1/100

number of temp	8	16	32	63
estimated area[mm^2]	0.692	0.664	0.608	0.615
energy value	471120	436638	430320	424478

3.6 Experimental Results and Discussions

The MCNC benchmark data [MCNC 1990], consisting of 125 cells and 147 nets, was chosen for our measurements. In the initial placement, the value of energy was 911520 and a lower bound of the chip area was estimated as 1.372[mm^2].

The PSA was executed in 20,000 inner loops, with exchanges every hundred inner loops. 64 processors can be used on Multi-PSI. The number of temperatures is 63, the highest temperature is 10,000, the lowest is 20 and other temperatures are determined proportionally. 5 temperatures are assigned to the lower temperature region. The lower bound of the area of the final solution is estimated as 0.615 [mm^2], reduced by 56.0 % in comparison to the initial solution. The execution time was about 30 minutes and the final energy was 424478. Table 3 shows the system performance of the relation between the number of temperatures and quality of solutions. When the number of temperatures is 32, 16 or 8, with the other conditions the same, the lower bounds of the final chip are estimated as shown in Table 3.

When the number of temperatures is 63, the cooling schedules adopted by the final solution were as follows. The initial temperature was 3823, the highest temperature in the process was 4487, and the number of temperatures the solution passed was 53. We observed that 10 solutions out of the initial 63 had been disposed of at the lowest temperature. This indicates that the mechanism of the automatic cooling schedules actually worked as intended.

When the number of temperatures is 8, the results are even worse. If the dispersions in energy for each temperature are too far from each other, the chance of exchange gets small. So the automatic cooling schedules will not work as intended. As a result, the algorithm can not get out of the *local optimum.*

To get an effective cooling schedule, it is necessary to

find the appropriate value of the highest temperature so that it can reach the disorder state. It is also necessary to adjust the number of temperatures according to the size of the problem.

As a future work, we are planning to study the mechanism for deciding the initial temperatures assigned to each processor from the energy dispersion of the solutions.

From the viewpoint of system performance, more speed-up and improvement in the ability to treat larger amounts of benchmark data are needed as the next step.

4 Logic Simulator

4.1 Background

The logic simulator is used in order to verify not only the functions of designed circuits but also the timing of signal propagation. Parallel logic simulation is treated as a typical application of Parallel Discrete Event Simulation (PDES). PDES can be modeled so that several objects (state automata) change their states by communicating with each other. A message has information on the event whose occurrence time is stamped on the message (time-stamp). In logic simulation, an object corresponds to a gate and an event means the change of the signal value.

In PDES, the time-keeping mechanism is essential for efficient execution. The mechanisms broadly fall into three categories: synchronous mechanisms, conservative mechanisms and optimistic mechanisms. Their peculiar shortcomings are widely known; the synchronous mechanisms require global synchronization, the conservative mechanisms often deadlock and the optimistic mechanisms need rollback.

We are targeting an efficient logic simulator on PIM, which is a distributed memory MIMD machine. We adopted an optimistic mechanism, the Time Warp mechanism (TW), whose rollback process has been considered to be heavy. In practice, however, TW has neither been evaluated in detail nor compared with other mechanisms on MIMD machines.

We expected that TW would be suitable for logic simulator on large-scale MIMD machines with some devices that reduced the rollback overhead. Thus a local message scheduler, an antimessage reduction mechanism and a load distribution scheme were added to our system and evaluated. Furthermore, we made two other simulators using different time-keeping mechanisms and compared the mechanisms with TW.

4.2 Time Warp Mechanism

The Time Warp mechanism[Jefferson 1985] was proposed by D. R. Jefferson. In PDES using TW, each object usually acts according to received messages and also records the history of messages and states, assuming that messages arrive chronologically. But when a message arrives at an object out of time-stamp order, the object rewinds its history (this process is called rollback), and makes adjustments as if the message had arrived in the correct time-stamp order. After rollback, ordinary computation is resumed. If there are messages which should not have been sent, the object also sends antimessages in order to cancel those messages.

4.3 System Specification

The system simulates combinatorial circuits and sequential circuits that have feedback loops. It handles three values: Hi, Lo, and X (unknown). A different delay time can be assigned to each gate (non-unit delay model). Since this simulator only treats gates, flip-flops and other functional blocks should be completely decomposed into gates.

4.4 Implementation

Since TW contains its peculiar overheads caused by the rollback processes, some devices for reducing overheads are needed for quick simulation. Furthermore, inter-PE communication overheads must be reduced because the simulator works on a distributed memory machine such as PIM.

For these purposes, a load distribution scheme, a local message scheduler and an antimessage reduction mechanism are included in our simulator. These are expected to reduce the overheads described above and might promote efficient execution of the simulator.

Each device is outlined below. Details are presented in [Matsumoto *et al.* 1992].

- Cascading-Oriented Partitioning

We propose the "Cascading-Oriented Partitioning" strategy for partitioning circuits to attain high-quality load distribution.

This scheme provides adequate partitioning solutions that satisfy these three requirements: load balancing, keeping inter-PE communication frequency low and deriving a lot of parallelism.

- Local Message Scheduler

During simulation, there are usually several messages to be evaluated in a PE. When the Time Warp mechanism is used, the bigger the time-stamp a message has, the more likely the message is to be rolled back. For this reason, appropriate message scheduling in each PE is needed for reducing rollback frequency.

- Antimessage Reduction

As long as messages are sent through the KL1 stream, messages arrive at their receiver in the same order as they are transmitted. In this environment, subsequent antimessages can be reduced. We adopted this optimization, expecting that it would reduce the rollback cost.

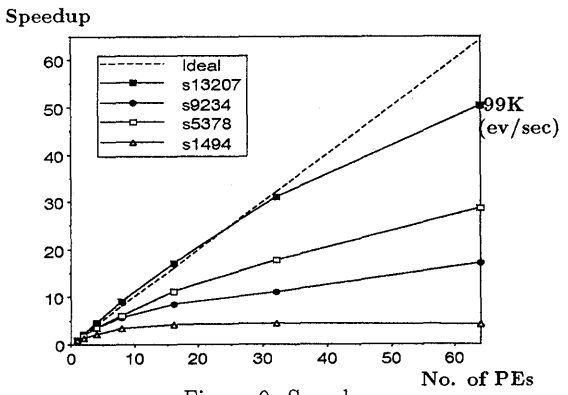


Figure 9: Speedup

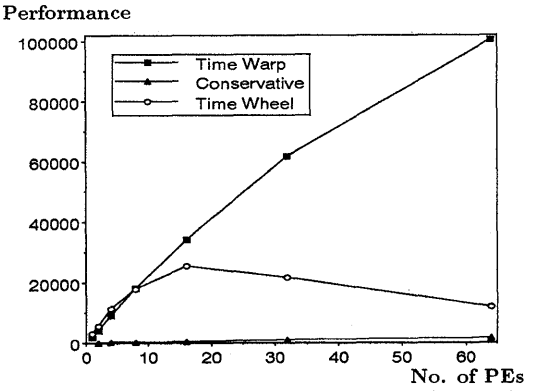


Figure 10: Performance Comparison (events/sec)

4.5 Measurements

We executed several experimental simulations on the Multi-PSI. Four sequential circuits, presented in ISCAS'89, were simulated in our experiments.

Figure 9 shows the system performance when the circuits were simulated using various numbers of PEs. The best performance is also shown there. In the best case, very good speedup of 48-fold was attained using 64 PEs. Approximately 99K events/sec performance, fairly good for a full-software logic simulator, was also attained.

4.6 Comparison between Time-keeping Mechanisms

For the purpose of comparing the Time Warp mechanism with others on the same machine, we made a further two simulators; one uses the synchronous mechanism and the other uses the conservative mechanism.

In the synchronous mechanism, only messages with the same time-stamp can be evaluated simultaneously. Therefore, a time wheel residing in each PE must synchronize globally at every tick. On the other hand, the problem of deadlock should be resolved [Misra 1986, Soulé *et al.* 1989] in conservative mechanisms. Our simulator basically uses null messages to avoid deadlock. A mechanism for reducing unnecessary null messages is also added in order to improve performance.

Figure 10 compares system performance when circuit s13207 was simulated under the same conditions (load distribution, input vectors, etc.).

The synchronous mechanism showed good performance using comparatively few PEs, however, the performance peaked at 16 PEs. Global synchronization at every tick apparently limits performance.

The conservative mechanism indicated good speedup but poor performance: using 64 PEs, only about 1.7 k events/sec performance was obtained. We measured the number of null messages generated during the simulation and found that the number of null messages was 40 times as many as that of actual events! That definitely was the cause of the poor performance.

This comparison substantiates that the Time Warp mechanism provides the most efficient simulation of the three mechanisms on distributed memory machines such as the Multi-PSI.

5 Concluding Remarks

This paper presented ICOT-developed parallel systems for routing, placement and logic simulation, and reported on their evaluation.

In the routing system, the router program was designed based on the concurrent objects model and congruent with the KL1 description was introduced. As a result, appreciably good speedup was attained and the quality of the solutions was high especially for large-scale data.

The parallel placement system is based on time-homogeneous SA, which realizes an automatic cooling schedule. The remarkable point of this system is that parallelization was applied not for the purpose of speedup, but to obtain high quality solutions.

The parallel logic simulator simply targeted quick execution. Absolutely good speedup was attained. The experimental results for three kinds of time-keeping mechanisms revealed that the Time Warp mechanism was the most efficient time-keeping mechanism on distributed memory machines.

These three systems are positive examples which support that PIM possesses high applicability to various practical problem domains as a general purpose parallel machine. Besides them, we are currently developing a hybrid layout system in which routing and placement are performed concurrently, improving interim solutions incrementally. These experiments, including the hybrid layout system, are just the preliminary experiments in the coming epoch of parallel machines, but they must be one of the most important and fundamental experiences for the future.

Acknowledgement

Valuable advice and suggestions were given by the members of PIC-WG, a working group in ICOT, during discussion of parallel LSI-CAD. The authors gratefully thank them. Data for the evaluation of our systems were recommended and given by NTT Co., Hitachi Ltd. and Fujitsu Ltd. We also thank these companies.

References

[Brouwer 1990] R. J. Brouwer and P. Banerjee. PHIGURE : A Parallel Hierarchical Global Router. In *Proc. 27th Design Automation Conf.*, 1990. pp. 650-653.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988. pp. 230-251.

[Fukui 1989] S. Fukui. Improvement of the Virtual Time Algorithm. *Transactions of Information Processing Society of Japan*, Vol.30, No.12 (1989), pp. 1547-1554. (in Japanese)

[Jefferson 1985] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Vol.7, No.3 (1985), pp. 404-425.

[Kawamura *et al.* 1990] K. Kawamura, T. Shindo, H. Miwatari and Y. Ohki. Touch and Cross Router. In *Proc. IEEE ICCAD90*, 1990. pp. 56-59.

[Kimura *et al.* 1991] K. Kimura and K. Taki. Time-homogeneous Parallel Annealing Algorithm. In *Proc. IMACS'91*, 1991. pp. 827-828.

[Kirkpatrick *et al.* 1983] S. Kirkpatrick, C. D. Gellat and M. P. Vecci. Optimization by Simulated Annealing, *Science*, Vol.220, No.4598, 1983. pp. 671-681.

[Kitazawa 1985] H. Kitazawa. A Line Search Algorithm with High Wireability For Custom VLSI Design, In *Proc. ISCAS'85*, 1985. pp. 1035-1038.

[Matsumoto *et al.* 1992] Y. Matsumoto and K. Taki. Parallel logic Simulator based on Time Warp and its Evaluation. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1992.

[MCNC 1990] *Proc. International Workshop Layout Synthesis '90* Research Triangle Park, North Carolina, USA, May 8-11, 1990.

[Misra 1986] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, Vol.18, No.1 (1986), pp. 39-64.

[Nair *et al.* 1982] R. Nair, S. J. Hong, S. Liles and R. Villani. Global Wiring on a Wire Routing Machine. In *Proc. 19th Design Automation Conf.*, 1982. pp. 224-231.

[Nakajima *et al.*1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2, In *Proc. 6th Int. Conf. on Logic Programming*, 1989. pp. 436-451.

[Olukotun *et al.* 1987] O. A. Olukotun and T. N. Mudge. A Preliminary Investigation into Parallel Routing on a Hypercube Computer, In *Proc. 24th Design Automation Conf.*, 1987. pp. 814-820.

[Rose 1988] J. Rose. Locusroute : A Parallel Global Router for Standard Cells, In *Proc. 25th Design Automation Conf.*, 1988. pp. 189-195.

[Sechen *et al.* 1985] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and Routing Package, *IEEE Journal of Solid-State Circuits*, Vol.SC-20, No.2, (1985), pp. 510-522.

[Soulé *et al.* 1989] L. Soulé and A. Gupta. Analysis of Parallelism and Deadlock in Distributed-Time Logic Simulation. *Stanford University Technical Report*, CSL-TR-89-378 (1989).

[Suzuki *et al.* 1986] K. Suzuki, Y. Matsunaga, M. Tachibana and T. Ohtsuki. A Hardware Maze Router with Application to Interactive Rip-up and Reroute. *IEEE Trans. on CAD*, Vol.CAD-5, No.4, (1986), pp. 466-476.

[Taki 1988] K. Taki. *The parallel software research and development tool: Multi-PSI system, Programming of Future Generation Computers*, pp. 411-426, North-Holland, 1988.

[Ueda *et al.* 1990] K. Ueda, T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol.33, No.6, (1990), pp. 494-500.

[Watanabe *et al.* 1987] T. Watanabe, H. Kitazawa, Y. Sugiyama. A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor. *IEEE Trans. on CAD*, Vol.CAD-6, No.2, (1987), pp. 241-250.

[Won *et al.* 1987] Y. Won, S. Sahni and Y. El-Ziq. A Hardware Accelerator for Maze Routing. In *Proc. 24th Design Automation Conf.*, 1987. pp. 800-806.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Database Management System: Kappa-P

Moto Kawamura Hiroyuki Sato †
Kazutomo Naganuma Kazumasa Yokota
Institute for New Generation Computer Technology (ICOT)
21F. Mita-Kokusai Bldg., 1-4-28 Mita, Minato-ku, Tokyo 108, Japan
e-mail: { kawamura, naganuma, kyokota }@icot.or.jp Tel: +81-3-3456-3069 Fax: +81-3-3456-1618

† Mitsubishi Electric Corporation
Computer & Information Systems Laboratory
5-1-1 Ofuna, Kamakura, Kanagawa 247, Japan
e-mail: hiroyuki@isl.melco.co.jp Tel: +81-467-46-3665 Fax: +81-467-44-9269

Abstract

A parallel database management system (DBMS) called *Kappa-P* has been developed in order to provide efficient database management facilities for knowledge information processing applications in the Japanese FGCS project. The data model of Kappa-P is based on a nested relational model for treating complex data structures, and has some new data types. Kappa-P has features of both a parallel DBMS on a tightly-coupled multiprocessor and a distributed DBMS on a loosely-coupled multiprocessor. In this paper, we describe the overview of Kappa-P.

1 Introduction

In the Japanese FGCS (Fifth Generation Computer System) project, many knowledge information processing systems (KIPSs) have been designed and developed under the framework of logic and parallelism. Among them, R&D of databases and knowledge-bases[14] aims at an integrated knowledge-base management system (KBMS) under a framework of deductive object-oriented databases (DOODs). *Kappa* [1] is a database management system (DBMS) located in the lower layer and is also a name of the project. The objective is to provide database management facilities for many KIPSs and support efficient processing of the FGCS prototype system as the database engine. In the Kappa project, we have developed a sequential DBMS, *Kappa-II* and a parallel DBMS, *Kappa-P*. Both systems adopt a nested relational model.

Kappa-II, which is a research result of the intermediate stage, is written in ESP, and works on sequential inference machines PSI and its operating system SIMPOS. The system showed us that our approaches based on the nested relational model are sufficient for KBMSs and KIPSs, and has been used as the DBMS on PSI machines by various KIPSs, for instance natural language processing systems with electronic dictionaries, proof checking systems with mathematical knowledge, and genetic information processing systems with molecular biological data.

A parallel DBMS project called Kappa-P[7] was initiated at the beginning of the final stage. Kappa-P is based on Kappa-II from a logical point of view, and its configuration and query processing have been extended for the parallel environment. Kappa-P is written in KL1 and works on the environment of PIM machines and their operating system PIMOS. The smallest configuration of Kappa-P is almost the same as Kappa-II. Compared both systems on the same machine, Kappa-P works with almost the same efficiency as Kappa-II. Kappa-P is expected to work on PIM more efficiently than Kappa-II, as their environments are different.

We describe the design policies in Section 2 and the features in Section 3. We explain the features of Kappa's nested relational model that are different from others in Section 4. Then, we describe an overview of the Kappa-P system: data placement in Section 5, management of global information in Section 6, query processing in Section 7, and implementation issues of element DBMSs in Section 8.

2 Design Policies

There are various data and knowledge with complex data structure in our environment. For example, molecular biological data treated by genetic information processing systems includes various kinds of information and huge amounts of sequence data. The GenBank/HGIR database[3] has a collection of nucleic acids sequences, the physical mapping data, and related bibliographic information. Amount of data has

[1] Knowledge Application-Oriented Advanced Database Management System

been increasing exponentially. Furthermore, the length of values is extremely variable. For example, the length of sequence data ranges from a few characters to 200,000 characters and becomes longer for genome data. Conventional relational model is not sufficient for efficient data representation and efficient query processing. Since the data is increasing rapidly, more processing power and more secondary memory will be required to manage it. Such situations require us to have a data model which can efficiently treat complex structured data and huge amount of data.

Parallel processing enables us to improve throughput, availability, and reliability. PIM-p is a hybrid MIMD multi-processor machine which has two aspects, a tightly-coupled multi-processor with a shared memory, called a *cluster*, and a loosely-coupled multiprocessor connected by communication networks. Disks can be connected to each cluster directly. The architecture can be that of typical PIMs. Both applications and Kappa-P are executed on the same machine. Both KBMSs and KIPSs need a lot of processing power to improve their response, so Kappa-P should be designed to improve the throughput. The system should use resources effectively, and be adapted for the environment.

For the above requirements, the system is designed as follows:

- In order to treat complex structured data efficiently, a nested relational model is adopted. The model is nearly the same as Kappa-II's data model, which shows us efficient handling of complex structured data. New data types and new indexed attributes should be added to handle huge amounts of data efficiently.

- The system should use system resources effectively to improve throughput. System resources are processing elements, shared memories, disks, and communication networks.

  - The system should use hybrid multiprocessors effectively.
  - Main memory database facilities should be provided for effective utilization of (shared) main memories. Because the data structure of the nested relation with variable occurrences and strings is complex, such a structure can be handled more efficiently on main memory than secondary memory.
  - The system should provide parallel disk access to reduce disk access overheads.
  - The system should actively control communication among clusters in order to reduce communication overheads in query processing.

- The system should be adapted for the environment. Though Kappa-P may be similar to database machines[1], the difference between Kappa-P and database machines is that both applications and a DBMS work together on the same machine.

  - The system should provide functions to reduce communication overheads between applications and the system, because they work together on the same machine. The functions execute part of applications at clusters which produce input data.
  - The system should provide a mechanism to process some queries in an application, because internal processing of an application is parallel, and some queries can occur in parallel.
  - The system uses the PIMOS file system, a part of which is designed for Kappa-P. The file system provides efficient access to large files, mirrored disks, and the sync mechanism for each file.

3 Features

According to the policies mentioned in the previous section, Kappa-P has been implemented. The system has the following features:

- Nested Relational Model

Already mentioned, conventional relational model is not appropriate in our environment. In order to treat complex structured data efficiently, a nested relational model is adopted. The nested relational model with a set constructor and hierarchical attributes can represent complex data naturally, and can avoid unnecessary division of relations. The model is nearly the same as Kappa-II's data model, which shows us efficient handling of complex structured data. Because Kappa-P is also the database engine for the KBMS of the FGCS project, the semantics of nested relations matches the knowledge representation language, *Quixote*[10] of the KBMS.

Term is added as a data type to store various knowledge. The character code of the PIM machine is based on 2-byte code, but the code wastes secondary memory space. In order to store a huge amount of data, such as a genome database in the near future, new data types and new indexed attributes are added.

- Configuration

The configuration of Kappa-P corresponds to the architecture of the PIM machine, and distinguishes

inter-cluster parallelism from intra-cluster parallelism. Kappa-P is constructed of a collection of *element DBMSs* located in clusters. These element DBMSs cooperate to process each other's queries.

Figure 1 shows the overall configuration of Kappa-P. The global map of relations is managed by some element DBMSs called *server DBMSs*. Sever DBMSs manage not only global map but also ordinary relations. Element DBMSs, with the exception of server DBMSs, are called *local DBMSs*. *Interface processes* are created to mediate application programs and Kappa-P, and receive queries as messages.

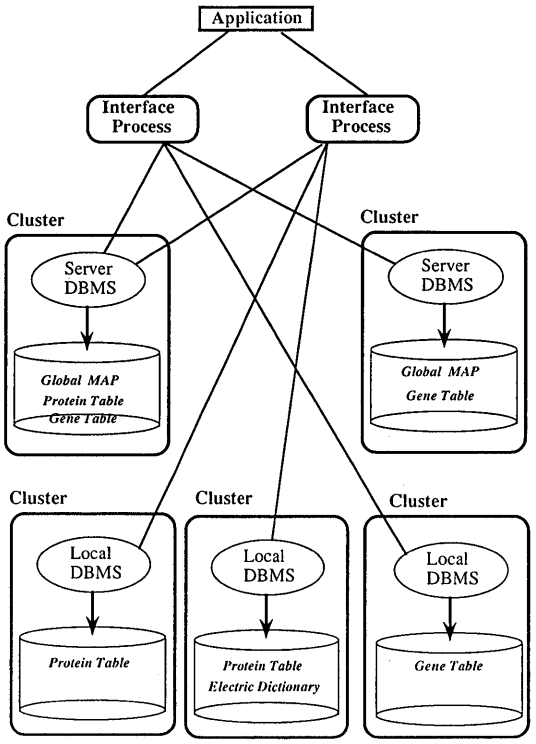


Figure 1: Configuration

- Data Placement

The placement of relations also corresponds to parallelism: inter element DBMS placement and intra element DBMS placement.

In order to use inter-cluster parallelism, relations can be located in some element DBMSs. The simple case is the distribution of relations like distributed DBMSs. When a relation needs a lot of processing power and higher bandwidth of disk access, the relation can be declustered as a horizontally partitioned relation and can be located in some element DBMSs. When a relation is frequently accessed in any query, some replicas of the relation can be made and can be located in some element DBMSs. However, in the current implementation, the replicated relation can be used for the global map only, that is, for server DBMSs.

Relations can be located in main memory or secondary memory in an element DBMS. Relations in main memory are temporary relations with no correspondent data in secondary memory. This means relation distribution in an element DBMS. A *quasi main memory database*, which guarantees to reflect those modifications to secondary memory, contains a relation in secondary memory and a replica of the relation in main memory.

- Query Processing

There are two kinds of commands for query processing: *primitive commands* and *KQL*, a query language based on extended relational algebra. Primitive commands are the lowest operations for relations, and can treat relations efficiently. The KQL is syntactically like KL1. New operations can be defined temporarily in a query.

A query in KQL is translated into sub-queries in intermediate operations for extended relational algebra, and is submitted to relevant element DBMSs. A query in primitive commands is submitted to relevant element DBMSs. The query is processed as a distributed transaction among relevant element DBMSs, and is finished under the control of two phase commitment protocol.

- Parallel Processing

Parallel processing of Kappa-P corresponds to the architecture of the PIM machine: inter-cluster parallelism among element DBMSs and intra-cluster parallelism in an element DBMS. The trade-off is processing power and communication overheads.

There are two kinds of parallel processing depending on data placement. Distribution of relations and horizontal partition of relations give us inter-cluster parallelism. In this case, a query is translated into sub-queries for some element DBMSs. Replication of a relation decentralizes access to the relation and improves availability.

- Compatibility to Kappa-II

The PIM machine is used via PSI machines acting as front-end processors. In order to use programs developed on PSI machines, such as terminal interfaces or application programs on Kappa-II, Kappa-P provides a program interface compatible to Kappa-II's primitive commands.

4 Nested Relational Model

A nested relational model is well known to reduce the number of relations in the case of multi-value dependency and to represent complex data structures more naturally than conventional relational model. However, there have been some nested relational models[8, 9, 2] since the proposal in 1978[6]. That is, even if they are syntactically the same, their semantics are not necessarily the same. In Kappa, one of the major problems is which semantics is appropriate for the many applications in our environment. Another problem is which part of *QUIXOTE* should be supported by Kappa as a database engine because enriched representation is a trade-off in efficient processing. In this section, we explain the semantics of the Kappa model.

Intuitively, a *nested relation* is defined as a subset of a Cartesian product of domains or other nested relations:

$$\begin{aligned} NR &\subseteq E_1 \times \cdots \times E_n \\ E_i &::= D \mid 2^{NR} \end{aligned}$$

where D is a set of atomic values [2]. That is, the relation may have a hierarchical structure and a set of other relations as a value. It corresponds to introducing tuple and set constructors as in complex objects. Corresponding to syntactical and semantical restrictions, there are various subclasses, in each of which extended relational algebra is defined.

In Kappa's nested relation, a set constructor is used only as an abbreviation for a set of normal relations as follows:

$$\begin{aligned} &\{r[l_1=a, l_2=\{b_1, \cdots, b_n\}]\} \\ &\quad \Leftrightarrow \quad \{r[l_1=a, l_2=b_1], \cdots, r[l_1=a, l_2=b_n]\} \end{aligned}$$

The operation of "$\Rightarrow$" corresponds to an *unnest* operation, while "$\Leftarrow$" corresponds to a *nest* or *group-by* operation. "$\Leftarrow$", however, is not necessarily congruent for the application sequence of nest or group-by operations. That is, in Kappa, the semantics of a nested relation is the same as the corresponding relation without set constructors. The reason why we take such semantics is to retain the first order semantics for efficient processing and to remain compatible with widely used relational model. Let a nested relation

$$\begin{aligned} &NR = \{nt_1, \cdots, nt_n\} \\ &\text{where } nt_i = \{t_{i1}, \cdots, t_{ik}\} \text{ for } i = 1, \cdots, n, \end{aligned}$$

then the semantics of NR is $\{t_{11}, \cdots, t_{1k}, \cdots, t_{n1}, \cdots, t_{nk}\}$. Extended relational algebra to this nested relational database is defined in Kappa and produces results according to the above semantics, which guarantees to produce the same result to the corresponding relational database, except treatment of attribute hierarchy.

As a relation among facts in a database is conjunctive from a proof-theoretic point of view, we consider a query as the first order language: that is, in the form of rules. The semantics of a rule constructed by nested tuples with the above semantics is rather simple. For example, the following rule

$$\begin{aligned} &r[l_1=X, l_2=\{a, b, c\}] \\ &\qquad \Leftarrow B, r'[l_3=Y, l_4=\{d, e\}, l_5=Z], B'. \end{aligned}$$

can be transformed into the following set of rules without set constructors:

$$\begin{aligned} &r[l_1=X, l_2=a] \Leftarrow \\ &\quad B, r'[l_3=Y, l_4=d, l_5=Z], r'[l_3=Y, l_4=e, l_5=Z], B'. \\ &r[l_1=X, l_2=b] \Leftarrow \\ &\quad B, r'[l_3=Y, l_4=d, l_5=Z], r'[l_3=Y, l_4=e, l_5=Z], B'. \\ &r[l_1=X, l_2=c] \Leftarrow \\ &\quad B, r'[l_3=Y, l_4=d, l_5=Z], r'[l_3=Y, l_4=e, l_5=Z], B'. \end{aligned}$$

That is, each rule can also be unnested into a set of rules without a set constructor. The point of efficient processing of Kappa relations is how to reduce the number of unnest and nest operations, that is, how to process sets directly.

Under the semantics, query processing to nested relations is different from conventional procedures. For example, consider a simple database consisting of only one tuple:

$$r[l_1 = \{a, b\}, l_2 = \{b, c\}].$$

For a query $?\text{-}r[l_1 = X, l_2 = X]$, we can get $X = \{b\}$, that is, an intersection of $\{a, b\}$ and $\{b, c\}$. That is, a concept of unification should be extended. In order to generalize such a procedure, we must introduce two concepts into the procedural semantics[11]:

1) *Residue Goals*
Consider the following program and a query:

$$\begin{aligned} &r[l=S'] \Leftarrow B. \\ &?\text{-}r[l=S]. \end{aligned}$$

If $S \cap S'$ is not an empty set during unification between $r[l = S]$ and $r[l = S']$, new subgoals are $r[l = S \setminus S'], B$. That is, a residue subgoal $r[l = S \setminus S']$ is generated if $S_1 \setminus S_2$ is not an empty set. Otherwise, the unification fails. Note that there might be residue subgoals if there are multiple set values.

2) *Binding as Constraint*
Consider the following database and a query:

$$\begin{aligned} &r_1[l_1=S_1]. \\ &r_2[l_2=S_2]. \\ &?\text{-}r_1[l_1=X], r_2[l_2=X]. \end{aligned}$$

[2] The term "atomic" does not carry its usual meaning. For example, when an atomic value has a type *term*, the equality must be based on unification or matching.

Although we can get $X = S_1$ by unification between $r_1[l_1 = X]$ and $r_1[l_1 = S_1]$ and a new subgoal $r_2[l_2 = S_1]$, the succeeding unification results in $r_2[l_2 = S_1 \cap S_2]$ and a residue subgoal $r_2[l_2 = S_1 \setminus S_2]$. Such a procedure is wrong, because we should have an answer $X = S_1 \cap S_2$. In order to avoid such a situation, the binding information is temporary and plays the role of the constraints to be retained.

The procedural semantics of extended relational algebra is defined based on the above concepts. According to the semantics, a Kappa database is allowed not necessarily to be *normalized* also in the sense of nested relational model, in principle: that is, it is unnecessary for users to be conscious of row nest structure. On the other hand, in order to develop deductive databases on Kappa, a logic programming language, called CRL [11] is developped on the semantics and is further extended in *Quixote* [10].

There remains one problem such that unique representation of a nested relation is not necessarily decided in the Kappa model, as already mentioned. In order to decide a unique representation, each nested relation has a sequence of attributes to be nested in Kappa.

Consider some examples of differences among some models. First, assume a relation r consisting of two tuples:

$$\{r[l_1 = a, l_2 = \{b, c\}], \\ r[l_1 = a, l_2 = \{c, d\}]\}$$

By applying a row-nest operation on l_2 to R, we get two possible relations:

$$\{r[l_1 = a, l_2 = \{b, c, d\}]\} \\ \{r[l_1 = a, l_2 = \{\{b, c\}, \{c, d\}\}]\}$$

According to the semantics of Kappa and Verso[9], we get the first relation, while, according to one of DASDBS[8] and AIM-P[2], we get the second.

Secondly, consider another relation r' consisting of only one tuple:

$$\{r'[l_1 = a, l_2 = \{b_1, b_2\}, l_3 = \{c_1, c_2\}]\}$$

By applying selection operations $\sigma_{l_2=b_1}$ and $\sigma_{l_3=c_1}$, we get the following two relations, respectively:

$$\{r'[l_1 = a, l_2 = b_1, l_3 = \{c_1, c_2\}]\} \\ \{r'[l_1 = a, l_2 = \{b_1, b_2\}, l_3 = c_1]\}$$

If we apply a union operation to the above two relations, we get two possible relations. According to the semantics of Verso, we get the following (original) relation:

$$\{r'[l_1 = a, l_2 = \{b_1, b_2\}, l_3 = \{c_1, c_2\}]\}.$$

That is, although the combination of $l_1 = a$, $l_2 = b_2$, and $l_3 = c_2$ is not selected after two selections, it comes back to life in the result of the union. On the other hand, according to the semantics of Kappa, we have one of the following:

$$\{r'[l_1 = a, l_2 = b_1, l_3 = \{c_1, c_2\}], \\ r'[l_1 = a, l_2 = b_2, l_3 = c_1]\}, \text{ or}$$

$$\{r'[l_1 = a, l_2 = \{b_1, b_2\}, l_3 = c_1], \\ r'[l_1 = a, l_2 = b_1, l_3 = c_2]\}$$

Which relation is selected depends on nested sequence defined in the schema.

According to the above semantics, the Kappa model guarantees more efficient processing by reducing the number of tuples and relations, and more efficient representation by the complex construction than relational model.

5 Data Placement

In order to obtain larger processing power using intercluster parallelism, relations should be located in different element DBMSs. Kappa-P provides three kinds of data placement: distribution, horizontal partition, and replication.

- Distribution

Distribution of relations is a simple case like distributed DBMSs. When relations are distributed in some element DBMSs, larger processing power can be obtained, but communication overheads may be generated at the same time. A database designer should be responsible for distribution of relations, because how to distribute relations relates to relationships among relations and kinds of typical queries to the database. In typical queries, strongly related relations should be in the same element DBMS, and loosely related relations might be in different element DBMSs.

A query to access these relations is divided into subqueries for some element DBMSs by an interface process (Figure 1), and each sub-query is processed as a distributed transaction.

- Horizontal Partition

A horizontally partitioned relation is a kind of declustered relation. It is logically one relation, but consists of some sub-relations containing distributed tuples according to some declustering criteria. A horizontally partitioned relation is effective when the relation needs a lot of processing power and higher bandwidth of disk access. For example, it is effective in a case of a molecular biological database which includes sequence data which requires homology search by a

pattern called motif. A database designer is also responsible for horizontal partition of relations, because horizontal partition does not always guarantees efficient processing if it does not satisfy declustering criteria.

A query to access horizontally partitioned relations is converted into sub-queries to access each sub-relation. Each sub-query is processed in parallel in a different clusters with sub-relations. Especially, when the query is a unary operation or a binary operation suitable for the declustering criteria, each sub-query can be processed independently and communication overheads among clusters can be disregarded. In other cases, as communication overheads among clusters can't be disregarded and it is necessary to convert the queries to reduce the overheads.

- Replication

Replication of a relation in some element DBMSs enables us to decentralize to access the relation, and to improve availability. Only the global map held in server DBMSs is replicated with a voting protocol in the current implementation of Kappa-P. The replication avoids centralizing access for server DBMSs, and even if some server DBMSs would stop, server facilities can work on.

6 Management of Global Information

Metadata of Kappa-P is divided into two kind of information: global information and local information. The global information consists of logical information, such as the database name and relation names, and physical information about element DBMSs, such as start-up information, current status, and stream to communicate. The local information also consists of logical information, such as the local database name and schema, and physical information, such as file names and physical structures of relations. Each element DBMS manages local information, and server DBMSs manage global information in addition ordinary relations. The role of server DBMSs is management of global information, especially, management of relation names for query processing and establishment of communication path between an interface process and element DBMSs.

- Management of Relation Names

It is necessary to guarantee the uniqueness of relation names. The simplest way is that a relation name forces to contain the relevant element DBMS name. Such a name is not suitable for Kappa-P, because Kappa-P treat logically one database. Server DBMSs manage relation names centrally, and provide location independent relation names. The information consists of a relation name and an element DBMS name in which the relation exists. This information is referred in order to find relevant element DBMSs from relation names at the beginning of query processing. When a relation is created, a message to register the relation name information is sent from the transaction. When a relation is deleted, a message to erase the relation name information is sent from the transaction.

Global information is replicated in order to decentralize ascesses to server DBMSs. Replication of the information is implemented by using a voting protocol. In order to access server DBMSs, a distributed transaction uses two phase commitment protocol.

- Management of Physical Information

Server DBMSs manage physical information, such as start-up information, current status, and stream to communicate. Sever DBMSs watch the state of element DBMSs. At the beginning of query processing, a server DBMS connects an interface process to relevant element DBMSs.

7 Query Processing

7.1 Query Language

There are two kinds of language for query processing: KQL, a query language based on extended relational algebra, and primitive commands. A query in both primitive commands and KQL is in the form of a message to an interface process, and the result is returned through the tuple stream which dose not have cursors as SQL.

- KQL(Kappa Query Language)

KQL is syntactically similar to KL1. Operations of extended relational algebra are written like predicates, and new operations can be defined temporarily, which take relations only as their arguments. Figure 2 shows a query in KQL.

- Primitive Commands

Primitive commands are the lowest operation for nested relations and a collection of unary operators for a nested relation. Figure 3 shows an example in primitive commands.

7.2 Query Processing

A query in KQL is processed in the following steps.

- Query Translation

```
go(Result::result1, Temp::result2) :- true |
     selection(table2, ''(from = "icot"), Temp),
     difference(table1, table1, EmptyTable),
     transitive_closure(table1, EmptyTable,
                table1, Result).

transitive_closure(Delta, In, R, Out) :-
     empty(Delta) |
     In = Out.

transitive_closure(Delta, In, R, Out) :- true |
     join(In, In, ''(to = from), In1),
     projection(In1, {'1.from', '2.to'},
            In2::{from, to}),
     union(In2, R, NextIn),
     difference(NextIn, In, Delta),
     transitive_closure(Delta, NextIn, R, Out).
```

Figure 2: Query in KQL

```
ifp:create(pc, off, IFP, Status0),
IFP  = [open([], Status1),
       begin_transaction([table1(read)],[],Status2),
       create_format(table1, ''(*), FMT, Status3),
       read_record(table1,FMT,rid,TupleStream0,S4)
     | IFP1],
TupleStream0 = [Buffer1 | TupleStream1],
               %% Buffer1 = [Tuple1,...TupleN]
TupleStream1 = [Buffer2 | TupleStream2],
               %% Buffer2 = [Tuple21,...Tuple2N]

IFP1 = [ end_transaction(Status5),
       close(Status6)].
```

Figure 3: Primitive Commands

A query in KQL is translated into sub-queries, which is called *intermediate operations* (shortly, *operations* in the following procedures) for extended relational algebra, by an interface process of Kappa-P.

1) Get relation names by parsing a query, and get location information of the relations from randomly selected server DBMSs.

2) Get schemata and supplementary information of the relations from relevant element DBMSs. In case of horizontally partitioned relations and quasi main memory relations, information of sub-relations is assembled into one. Supplementary information is followings:

   - List of indexed attributes
     The algorithm of query processing is dependent on whether an attribute is indexed or not.
   - Uniqueness of attribute value
     The algorithm is dependent on whether an attribute value is unique or not.
   - Kinds of attribute values and the number of attribute values
     The information is used to estimate amounts of intermediate results, and reduce communication costs.
   - The number of tuples and the average size of tuples
     The information is also used to estimate amounts of intermediate results.

3) Replace an operation for a horizontally partitioned relation with some operations for sub-relations and add merge operations. In case of a quasi main memory relation, replace an update operation with operations both the secondary memory relation and the temporary relation. Replace a non-update operation with an operation for the temporary relation.

4) The executing order of operation in the query is extracted from the query, and operations to control executing order are embedding in the query. Since the query can include update operations, it is impossible to control the data flow graph only.

5) Using basic optimization techniques by supplementary information of relations, the query is translated, and an algorithm for processing extended relational algebra is determined. In this phase, some execution plans are produced.

6) According to the location information of relations, the candidates are divided into sub-sequences to minimize the communication costs estimated by the supplementary information. Sub-sequences with the least communication cost are chosen, and operations to transfer tuples are embedded in the sub-sequences.

7) Each sub-sequence is translated into KL1 program with procedures calling intermediate operations.

- Query Execution

Each sub-sequence is sent to the related element DBMS, and processed. Each sub-sequence is executed as a distributed transaction with two phase commitment protocol. Although processing in an element DBMS is based on tuple streams, data in other element DBMSs are accessed via transfer operations embedded in the query translation phase.

7.3 User Process in Element DBMS

Because both Kappa-P and application programs work together on the same machine, the system cannot only provide higher communication bandwidth between them, but can also reduce communication overheads between them by allocating them in the same cluster.

In primitive commands, a tuple filter are taken as the argument of a read operation. The read operation invocates the filter in the same cluster in which a relation exists. If the relation is horizontally partitioned, filters for each sub-relation is invocated, and the outputs of all filters are merged into one.

In KQL, a filter is specified as one of the new operations.

8 Element DBMS

An element DBMS contains full database management facilities, and accepts intermediate operations for extended relational algebra and primitive commands. We are not concerned with communication overheads in element DBMSs. Kappa-P uses parallel processing on a shared memory only, but doesn't use parallel operations for secondary memory in element DBMSs, in the current implementation.

- Parallel Processing by Tuple Stream

Stream programming is a very typical programming style in KL1. In general, a query can be expressed as a graph, which consists of some nodes corresponding to the operations and arcs corresponding to relationships among operations. In KL1, the graph corresponds to the processing structure of the query. The nodes become processes, and arcs become streams through which tuples are sent. In KL1, the number of tuples in the streams does not only depend on the amount of intermediate results, but also the number of processes to be scheduled. So, it is very important to control the number of tuples, and to drive the streams on demand with double buffering.

Figure 4 shows an example for parallel processing by tuple stream: $Table\,3 = \pi_{[a,b]} Table\,1 \bowtie Table\,2$.

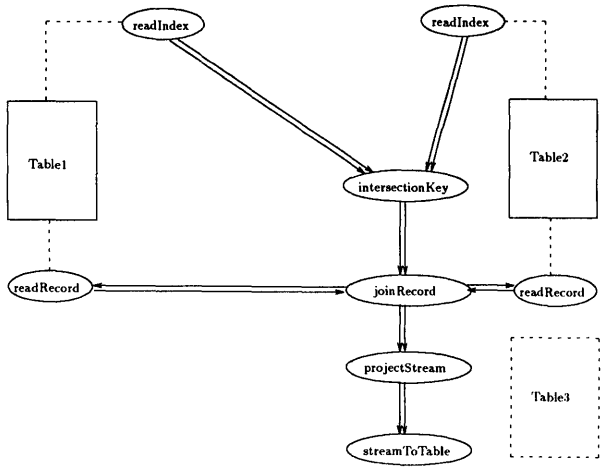


Figure 4: Parallel Processing by Tuple Stream

- Parallel Processing of Primitive Commands

Primitive commands process various operations in parallel for nested relations, for instance, operations set for and index operations of temporary relations.

A set is a collection of tuple identifiers, and is obtained by restriction operations. In order to parallelize set operations, a set is partitioned according to the range of tuple identifiers.

Index structure of temporary relations is T-tree[5], which is more sufficient in main memory than B-tree. Range retrieval operations, we are processed in parallel. In general, leaf nodes are connected in order like B^+-tree to trace succeeding leaf node directory. In our experiments, such a structure can't work efficiently in KL1. Range retrieving on a tree, whose leaf nodes are connected, is done following steps: finding the minimum value of the range, and then, tracing through the connection until the maximum value is found.. These steps are almost processed sequentially. Assuming that H is the height of the tree, and R is the number of leaf nodes between the range, the number of comparison of values is $H + R$. On the other hand, range retrieving on a tree whose leaf nodes are not connected is done following steps: finding a minimum value of the range, finding a maximum value of the range, and collecting values between them. These steps are almost processed in parallel. The number of comparison is $2H$. The latter has advantages about parallelism, wide range retrieving, and efficient implementation in KL1.

- Main Memory Database Facilities

Each cluster of PIM has hundreds of mega bytes of main memory. In order to use such a large memory effectively, Kappa-P provides temporary relations and quasi main memory database facilities. Because tuples of nested relations with variable occurrences and strings are complex, such a structure can be handled more efficiently in main memory than in secondary memory.

Temporary relations exist only in main memory having no correspondent data in secondary memory, so modifications to the temporary relations are not reflected to secondary memory. But, temporary relations are useful for application programs which create many intermediate relations such as deductive databases. The temporary relations have the same interface as secondary memory relations.

A quasi main memory database, which guarantees to reflect those modifications to secondary memory, is not a pure main memory database, but parallel processing enables the quasi main memory database to work with nearly the same throughput as a main memory database. A quasi main memory relation is a kind of

replicated relations consisting of a pair of a secondary memory relation and a temporary relation. Kappa-P guarantees that both relations have the same logical structure, such as tuples, indexed attributes, and the same tuple identifiers, even if the relation is updated. Operations except for update operations can be executed by the temporary relation, because the temporary relation is processed faster than the secondary memory relation. Update operations should be executed by relations in parallel and asynchronously, and synchronization is achieved by two phase commitment protocol, which guarantees the equivalence of their contents.

9 Conclusions

In this paper, we described a parallel DBMS Kappa-P.

In order to provide KBMSs and KIPSs with efficient database management facilities, the system adopts a nested relational model, and is designed to use parallel resources efficiently by using various parallel processing. The smallest configuration of Kappa-P is almost the same as Kappa-II. Compared both systems on the same machine, Kappa-P works with almost same efficiency as Kappa-II. Kappa-P is expected to work on PIM more efficiently than Kappa-I. We will make various experiments for efficient utilization of parallel resources, and show that the system provides KBMSs and KIPSs with efficient database management facilities in the FGCS prototype system.

Acknowledgment

The Kappa project has had many important contributions in addition to the listed authors. The members of biological databases of the third research laboratory: Hidetoshi Tanaka and Yukihiko Abiru, the users of Kappa-II, and Kaoru Yoshida of LBL have shown us many suggestions for improvements. We thank Hideki Yasukawa of the third research laboratory for useful suggestions. The authors are grateful to Kazuhiro Fuchi and Shunichi Uchida for encouraging the projects.

References

[1] D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of Database Processing or a Passing Fad ?", *SIGMOD RECORD*, Vol.19, No.4, Dec.,1990.

[2] P. Dadam, et al, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies", *Proc. SIGMOD*, 1986.

[3] "GenBank/HGIR Technical Manual", *LA-UR 88-3038*, Group T-10, MS-K710, Los Alamos National Laboratory, 1988.

[4] M. Kawamura and H. Sato, "Query Processing for Parallel Database Management System", *Proc. KL1 Programming Workshop '91*, 1991. (in Japanese)

[5] T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems", *Proc. VLDB*, 1986.

[6] A. Makinouchi, "A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model", *Proc. VLDB*, 1977.

[7] H. Sato and M. Kawamura, "Towards a Parallel Database Management System (Extended Abstract)", *Proc. Joint American-Japanese Workshop on Parallel Knowledge Systems and Logic Programming*, Tokyo, Sep. 18-20, 1990.

[8] H.-J. Schek and G. Weikum, "DASDBS: Concepts and Architecture of a Database System for Advanced Applications", *Tech. Univ. of Darmstadt, TR*, DVSI-1986-T1, 1986.

[9] J. Verso, "VERSO: A Data Base Machine Based on Non 1NF Relations", *INRIA-TR*, 523, 1986.

[10] H. Yasukawa, H. Tsuda, and K. Yokota, "Object, Properties, and Modules in *Quixote*", *Proc. FGCS'92*, Tokyo, June 1-5, 1992.

[11] K. Yokota, "Deductive Approach for Nested Relations", *Programming of Future Generation Computers II*, eds. by K. Fuchi and L. Kott, North-Holland, 1988.

[12] K. Yokota, M. Kawamura, and A. Kanaegami, "Overview of the Knowledge Base Management System (KAPPA)", *Proc. FGCS'88*, Tokyo, Nov.28-Dec.2, 1988.

[13] K. Yokota and S. Nishio, "Towards Integration of Deductive Databases and Object-Oriented Databases — A Limited Survey", *Proc. Advanced Database System Symposium*, Kyoto, Dec., 1989.

[14] K. Yokota and H. Yasukawa, "Towards an Integrated Knowledge-Base Management System — Overview of R&D for Databases and Knowledge-Bases in the FGCS project", *Proc. FGCS'92*, Tokyo, June 1-5, 1992.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Objects, Properties, and Modules in *Quixote*

Hideki Yasukawa, Hiroshi Tsuda, and Kazumasa Yokota

Institute for New Generation Computer Technology (ICOT)
21F. Mita-Kokusai Bldg., 1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

e-mail: yasukawa@icot.or.jp, tsuda@icot.or.jp, kyokota@icot.or.jp

Abstract

This paper describes a knowledge representation language *Quixote*. *Quixote* is designed and developed at ICOT to support wide range of applications in the Japanese FGCS project.

Quixote is basically a deductive system equipped with the facilities for representing various kinds of knowledge, and for classifying knowledge.

In *Quixote* , basic notions for representing concepts and knowledge are *objects* and their *properties*. Objects are represented by extended terms called *object terms*, and their properties are represented by *subsumption* constraints over the domain of object terms.

Another distinguished feature of *Quixote* is its concept of *modules*. Modules play an important role in classifying knowledge, modularizing a program or a database, assumption-based reasoning, and so on.

In this paper, the concepts of objects, properties, and modules are presented. We also present how modules work with objects and their properties.

1 Introduction

Logic programming is a powerful paradigm for knowledge information processing systems from the viewpoint of knowledge representation, inference, advanced databases, and so on.

Quixote is designed and developed to support this wide range of applications in the Japanese FGCS project. Briefly speaking, it is a constraint logic programming language, a knowledge representation language, and a *deductive object-oriented database* language.

In *Quixote*, basic notions for representing concepts and knowledge are *objects* and their *properties*. An object in *Quixote* is represented by an extended term called an *object term*, and its properties are defined as a set of *subsumption* constraints.

Another distinguished feature of *Quixote* is its concept of *modules*. A module corresponds to a part of the world (situation) or the local database. In *Quixote*, its module concepts play an important role in classifying knowledge, modularizing a program or a database, assumption-based reasoning, and so on.

In this paper, concepts of objects, properties, and modules are presented. We also present how modules work with objects and their properties, for example, in classifying or modularizing them.

Other features of *Quixote* and the formalism appear in other papers[20, 12, 21].

Section 2 shows how objects and their properties are treated in a simple version of *Quixote*. Section 3 shows how complex objects are introduced in *Quixote*, and how they are used to deal with *exceptions* in *property inheritance*. Section 4 describes deductive rules in *Quixote*, and the overview of deductive aspects of *Quixote*. Section 5 describes module concepts with some examples. Section 6 describes the facilities for relating modules, especially to import or to export rules among modules. Section 7 describes queries in *Quixote*, which provides the facilities to deal with modifications of a program, or assumption-based reasoning. Finally, Section 8 describes a brief comparison with related works.

2 A simple system of objects and their properties

Object-oriented features are very useful for applying logic programming to 'real' applications. *Quixote* is designed as a logic programming language with features such as: object identity, complex objects, encapsulation, inheritance, and methods, which are also appropriate for deductive object-oriented databases and situation theoretic approaches to natural language processing systems.

An *object* is a key feature in *Quixote* to represent concepts and knowledge. In knowledge representation applications, it is important to identify an object or to distinguish two objects, as in the case of object-oriented languages.

Object identity is the basic notion for identifying objects.

Quixote precisely defines object identity, where extended terms are used as *object identifiers*. In this sense, extended terms in *Quixote* are called *object terms*.

In this section, the simplified treatment of objects and their properties are presented. That is, the case of every

object term is *atomic*. In the next section, the system of object terms is extended to non-atomic and complex cases, including the non-well-founded (circular) case.

2.1 Basic Objects

At the first approximation, we assume that each object has a unique atomic symbol as its identifier.

The important thing, here, is that objects are related to each other. There are some relations to be considered, such as *is_a*-relations, *part_of*-relations, and so forth. In *Quixote*, *subsumption* relations among objects are used to relate objects.

A set BO of atomic symbols called *basic objects* is assumed. BO is partially ordered by the *subsumption* relation (written $\sqsubseteq$), and $(BO, \sqsubseteq, \top, \bot)$ is a lattice with $\top$ as its maximum element and $\bot$ as its minimum element.

A basic object is used as an object identifier (an object term) in this simple setting.

An example of the lattice is

$$BO^* = (\{animal, mammal, human, dog\}, \sqsubseteq, \top, \bot)$$

where the following holds:

$$mammal \sqsubseteq animal,$$
$$human \sqsubseteq mammal,$$
$$dog \sqsubseteq mammal.$$

2.2 Attribute Terms and Dotted Terms

In addition to the basic objects, we assume a subset L of BO, called *labels*. Labels are used to define the attributes of objects.

An attribute of the object o is represented by the triple (o, l, v) where l is a label and v is an object. The following example shows that John has the attribute of his age being 20: $(john, age, 20)$.

A property of an object is represented by a set of the pairs of a label and its value, that is, a set of attributes. Thus, John's having a property of being 20 years old and being a male is represented by $\{(john, age, 20), (john, sex, male)\}$.

Formally, a label l is interpreted as a function:

$$[\![l]\!] : BO \rightarrow BO.$$

The syntactic construct for representing an object and its properties is the *attribute term*. An attribute term is of the form:

$$o/[l_1 = v_1, \ldots, l_n = v_n]$$

where o is an object term, l_i's are labels, and v_i's are objects. The syntactic entity $[l_1 = v_1, \ldots, l_n = v_n]$ is called the *attribution* of the object term o. It specifies a property of the object. In what follows, we say that o has the attributes $[l_1 = v_1, \ldots, l_n = v_n]$ when there is no confusion.

For example, the following is an attribute term representing that John has the property of being 20 years old and being a male:

$$john/[age = 20, sex = male].$$

Notice that an object identifier and its property (attribution) are separated by "/".

It is useful to regard an attribute of an object as a concept. For example, John's age can be seen as a concept. In *Quixote*, this kind of concept is represented by *dotted terms*. A dotted term is defined as a pair of an object term and a label, and has the following form:

$$o.l$$

where o is an object term and l is a label.

For example, John's age is represented by the following dotted term:

$$john.age = 20.$$

A dotted term is treated as a *global variable* ranging over the domain of object terms, and interpreted as an object term. The following holds for dotted terms:

$$o_1 = o_2 \Rightarrow o_1.l = o_2.l$$
$$o.l = o_1, o.l = o_2 \Rightarrow o_1 = o_2$$
$$o.l_1 = o_1 \Rightarrow o.l_1.l_2 = o_1.l_2.$$

2.3 Properties as Subsumption Constraints

It is often the case that an object has certain attribute while its value is not fully specified.

$$john/[age \rightarrow positive\_integer]$$

The above attribute term represents that John has the property of his age being subsumed by positive integer. In this case, John's age is not specified but constrained as being subsumed by *positive_integer*.

Constraints in the simplified *Quixote* are subsumption constraints over basic objects. As mentioned in 2.1, the domain of basic objects is a lattice under the subsumption relation. Thus, the rules of subsumption constraints are simply defined as follows:

- $x = x$,
- if $x = y$ then $y = x$,
- if $x = y$ and $y = z$ then $x = z$,
- $x \sqsubseteq x$,

- if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$,
- if $x = y$ and $x \sqsubseteq z$ then $y \sqsubseteq z$,
- if $x = y$ and $z \sqsubseteq x$ then $z \sqsubseteq y$,
- if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$,
- if $x \sqsubseteq y$ and $x \sqsubseteq z$ then $x \sqsubseteq (y \downarrow z)$, and
- if $y \sqsubseteq x$ and $z \sqsubseteq x$ then $(y \uparrow z) \sqsubseteq x$

where $(x \downarrow y)$ is the infimum (meet) and $(x \uparrow y)$ is the supremum (join). Note that $x = y$ is equivalent to the conjunction of $x \sqsubseteq y$ and $y \sqsubseteq x$, that is, the following holds:

$$x = y \stackrel{def}{\equiv} x \sqsubseteq y \wedge x \sqsupseteq y.$$

A set of subsumption constraints is solvable if and only if it does not contain $a = b$ for two distinct basic objects a and b with respect to the above rules[20].

The aforementioned attribute term is defined as a pair of the basic object *john* and a subsumption constraint $john.age \sqsubseteq positive\_integer$. Such a pair can also be written as:

$$john/|\{john.age \sqsubseteq positive\_integer\}.$$

This is just the opposite of the description of the attribute term shown above.

The general form of an attribute term is as follows:

$$o/[l_1\ op_1\ v_1, \ldots, l_n\ op_n\ v_n]$$

where $op_i \in \{=, \rightarrow, \leftarrow\}$.

For each label not explicitly specified, we assume that its value is constrained, that is, subsumed by $\top$. This assumption states that the property of an object can be partially specified.

In addition to the subsumption constraints over basic objects, the subsumption constraints over sets of basic objects are also used in QUIXOTE. For example, the following attribute term represents that both cooking and walking are John's hobbies:

$$john/[hobby \leftarrow \{cooking, walking\}],$$

that is, the dotted term *john.hobby* is a set subsuming the set $\{cooking, walking\}$.

The subsumption relation $\sqsubseteq_H$ over the domain of sets of basic objects is defined as Hoare-ordering over $\sqsubseteq$-ordering as follows:

$$s_1 \sqsubseteq_H s_2 \Rightarrow \forall x \in s_1 \exists y \in s_2\ x \sqsubseteq y.$$

It should be noted that the domain of sets of basic objects is classified by the equivalence relation defined by the Hoare-ordering. In QUIXOTE, any set is interpreted as the representative element of the equivalence class that is defined by the following rule:

$$[\![s]\!] \stackrel{def}{=} \{x \in s \mid \neg \exists y \in s\ x \neq y \wedge x \sqsubseteq y\}.$$

Under this definition, $\sqsubseteq_H$-ordering becomes a partial ordering, and can be used as an equivalence relation.

For example, the following holds:

$$[\![\{1, integer, \text{“}abc\text{”}, string\}]\!] = \{integer, string\},$$

provided that $1 \sqsubseteq integer$ and $\text{“}abc\text{”} \sqsubseteq string$.

2.4 Property Inheritance

It is natural to assume that properties are inherited from object terms with respect to $\sqsubseteq$-ordering. Consider the following example:

$$swallow \sqsubseteq bird.$$
$$bird/[canfly \rightarrow yes].$$

Since *swallow* is a kind of (subsumes) *bird* and *bird* has the attribute $[canfly \rightarrow yes]$, *swallow* has the same attribute by default.

The rule for inheritance of properties between objects is:

Definition 1 (Rule for inheritance)

$$o_1 \sqsubseteq o_2 \Rightarrow o_1.l \sqsubseteq o_2.l.$$

If $o_1 \sqsubseteq o_2$ holds, then the following holds according to this rule:

- if o_2 has the attribute $[l \rightarrow o']$, then o_1 also has the same attribute,
- if o_1 has the attribute $[l \leftarrow o']$, then o_2 also has the same attribute.

Notice that the attribute $[l = o']$ is the conjunction of $[l \rightarrow o']$ and $[l \leftarrow o']$.

As mentioned before, an attribute term consists of an object term and a set of subsumption constraints, thus, property inheritance can be considered as constraint inheritance.

3 Complex Objects

The simplified approach shown in the previous section lacks the capability to represent the complex objects required in actual applications, such as trees, graphs, proteins, chemical reactions, and so forth.

A complex object has certain "structures" intrinsic to its nature. Knowledge representation languages must be able to represent such complex structures, that is, the object identifiers in QUIXOTE language.

Thus, it is important to give a facility for introducing complex object terms into QUIXOTE.

3.1 Intrinsic vs. Extrinsic Properties

The approach adopted in *Quixote* is a natural extension of the simplified language given in the previous section.

An object has the property, that is, a set of attributes, which are *intrinsic* to identifying that object. Thus, the properties of an object are separated into two, the intrinsic property and the other extrinsic properties. Similarly, the attributes of an object are divided into two, intrinsic attributes and extrinsic attributes. In *Quixote*, the intrinsic attributes are included in the object term representation but not in the attribution of an attribute term representation.

For example, the concept of *red apple* is represented by the following complex object term:

$$apple[color = red].$$

Notice the difference between this object term and the attribute term $apple/[color = red]$. The latter represents the concept of *apple* with the attribute $[color = red]$ as its extrinsic property.

Let o be a basic object, $l_1, l_2, \ldots$ be labels, and $o_1, o_2, \ldots$ be object terms.

- Every basic object is an object term.
- A term $o[l_1 = o_1, l_2 = o_2, \ldots]$ is an object term if it contains only one value specification for each label.
- A term is an object term only if it can be shown to be an object term by the above definition.

For an object term $o[l_1 = o_1, l_2 = o_2, \ldots]$, o is called the *principal object* and $[l_1 = o_1, l_2 = o_2, \ldots]$ is called the *intrinsic property specification.* The intrinsic property specification of an object term is the set of intrinsic attributes of the object term, and interpreted as the indexed set of object terms indexed by the labels. Thus, an object term is interpreted as the pair of its principal object o and the indexed set s, and is written as:

$$(o, s).$$

Let $BO = \{human, 20, 30, int, male, female\}$, $20 \sqsubseteq int, 30 \sqsubseteq int$, $L = \{age, sex\}$. The following terms are object terms in *Quixote*:

$$human,$$
$$human[age = 20, sex = male].$$

These two object terms are interpreted as $(human, \{\})$ and $(human, \{(age, 20), (sex, male)\})$.

The object term $\top[l_1 = v_1, \ldots]$ is described as $[l_1 = v_1, \ldots]$ for convenience.

By the definition of complex object terms, the following holds:

$$o[\ldots, l_i = v_i, \ldots].l_i = v_i.$$

For example, $human[age = 20].age = 20$ holds.

It is possible to have object terms containing variables ranging over ground object terms as follows:

$$human$$
$$human[age = X, sex = Y].$$

3.2 Extended Subsumption Relation

Given the subsumption relations $\sqsubseteq$ among basic objects, the relations can be extended into subsumption relations among complex object terms. The extended subsumption relations preserve the ordering on basic objects, and also constitute a lattice.

The precise definition of a extended subsumption relation is given in [20], intuitive understanding will suffice at this point. Intuitively, $o_1 \sqsubseteq o_2$ (we say o_2 subsumes o_1) holds between two complex object terms o_1 and o_2 if and only if:

(1) the principal object of o_2 subsumes the principal object of o_1,

(2) o_1 has more labels than o_2, and

(3) the value of each label of o_2 subsumes the value of each label of o_1.

For example, the following holds:

$$human[age = 20, sex = male] \sqsubseteq animal[age = integer],$$

because the principal object of $animal[age = integer]$ $(animal)$ subsumes the principal object of $human[age = 20, sex = male]$ $(human)$, the object term $human[age = 20, sex = male]$ has more labels than $animal[age = integer]$, and $20 \sqsubseteq integer$ holds.

Similarly,

$$human[age = 20] \sqsubseteq animal[age = int]$$

holds, but $human[age = 20]$ and $human[sex = male]$ cannot be compared with respect to $\sqsubseteq$-ordering over complex object terms.

In such extended subsumption relations over object terms, the object term $\top$ is the largest among all the object terms. In *Quixote*, the object term $\bot$ is the smallest of all, that is, $\bot$ is used as the representative element of the class of object terms that are smaller than $\bot$[1].

The semantic domain of object terms is a set of labeled graphs, a subclass of hypersets with urelement[2, 13].

[1] From the definition of object terms and subsumption relation over them, it is possible to have an object term of the form:

$$\bot[l_1 = v_1, \ldots].$$

The reason such a domain is adopted is to allow object terms with infinite structure. Subsumption relations correspond to *hereditary subset relations*[2] on that domain.

The rules for extended subsumption constraints are those listed in 2.3 plus the following:

- if $(o_1, s_1) \sqsubseteq (o_2, s_2)$ then $o_1 \sqsubseteq o_2$ and for each $(l, v_2) \in s_2$ there exists (l, v_1) such that $v_1 \sqsubseteq v_2$,
- if $(o_1, s_1) = (o_2, s_2)$, then $o_1 = o_2$ and for each $(l, v_1) \in s_1$ there exists $(l, v_2) \in s_2$ such that $v_1 = v_2$ (the symmetric condition follows).

These two rules correspond to the *simulation* and *bisimulation* relations in [2, 13], where the bisimulation relation is an equivalence relation.

3.3 Exception on Property Inheritance

By introducing complex object terms in terms of intrinsic-extrinsic distinction, it becomes possible to define the notion of *exceptions* on the inheritance of properties in a clear way.

Intuitively, the intrinsic property of an object is the property that distinguishes that object from others, and such properties should not be inherited.

In addition to the rule for property inheritance given in 2.4, the rule for exception is defined as follows:

Definition 2 (Rule for exception)
The intrinsic attributes of an object term override the attribution inherited from the other object terms, and any of the intrinsic attributes is not inherited to the other object terms.

In sum, the intrinsic attributes are out of the scope of property inheritance.

For example, consider the attribute of the object term $bird[canfly \rightarrow no]$ with respect to the following database definition:

$$bird/[canfly = yes],$$
$$bird[canfly = no].$$

The object term $bird[canfly = no]$ inherits the attribute $[canfly \rightarrow yes]$, by the rule for inheritance. However, $bird[canfly = no]$ contains the intrinsic specification on the label $canfly$. Thus, $bird[canfly = no]$ has the attribute $[canfly \rightarrow no]$ as its property by the rule for exception.

Thus, given in Section 3.1, the following holds even if property inheritance occurs:

$$o[l_1 = x_1, \ldots, l_n = x_n]/[l_1 = x_1, \ldots, l_n = x_n].$$

4 Deductive Rules

It is important for knowledge representation languages to provide facilities for certain types of inferences, namely, deductive inference.

The deductive system of $\mathcal{QUIXOTE}$ is defined by *deductive rules* (rules, for short).

4.1 Rules in $\mathcal{QUIXOTE}$

First, a literal (atomic formula) of $\mathcal{QUIXOTE}$ is defined to be an object term or an attribute term.

The rules of $\mathcal{QUIXOTE}$ are defined as follows:

(1) a literal H,

(2) $H \Leftarrow B_1, \ldots, B_n$ where $H, B_1, \ldots, B_n$ are literals.

H is called the *head* and the "$B_1, \ldots, B_n$" is called the *body* of the rule.

Rules of the form (1) are sometimes called *unit rules* or *facts*[2].

Rules of the form (2) are called *non-unit rules.*

A fact H is shorthand for the non-unit rule whose body is empty, that is, the rule $H \Leftarrow$. When there is no confusion, non-unit clauses are simply called rules.

A *database* or a *program* is defined as a finite set of rules.

A fact specifies the existence of an object and its property. The following is an example of facts:

$$john;;$$
$$john/[age = 20];;$$

The former fact specifies that the literal $john$ holds (or is true), that is, the database has the object $john$ as its member. In addition to that, the latter specifies that $john$ has the property of $[age = 20]$.

The informal meaning of the rule $H \Leftarrow B_1, \ldots, B_n$ is as usual, that is, if $B_1, \ldots, B_n$ holds then H holds.

As mentioned in Section 2.3, properties are interpreted as subsumption constraints. Thus, a rule is defined as a triple (H, B, C) of the object term H in its head, the set of object terms B in its body, and the set of constraints C. The elements of B are called *subgoals.* Thus, any rule can be represented by the following form:

$$H \Leftarrow B \parallel C.$$

This form of rule is called *constraint-based* form.

It is possible to associate constraints, other than those corresponding to attributes, with a rule as follows:

$$john/[daughter \leftarrow \{X\}] \Leftarrow$$
$$X/[father = john] \parallel \{X \sqsubseteq female\}.$$

[2]Sometimes, a fact is defined to be a unit-rule having a *non-parametric* object term as its head. In that case, the set of facts corresponds to an extensional database in conventional deductive databases.

Precisely speaking, the set of constraints C of a rule is classified into two, the constraints in the head of the rule (*head constraints*) and the constraints in the body (*body constraints*). For example, the rule

$$o/[l_1 = o_1, l_2 = o_2] \Leftarrow \\ p/[l_3 = o_3], q/[l_4 = o_4]$$

has $\{o.l_1 = o_1, o.l_2 = o_2\}$ as its head constraints, and $\{p.l_3 = o_3, q.l_4 = o_4\}$ as its body constraints.

In the context of object-oriented languages, the attributes in the head of a rule correspond to the *methods*, and the body of the rule corresponds to their *implementation*, as in F-logic[8].

4.2 Derivations and Answers

Compared to the usual notion of the derivation of goals and answers in logic programming languages like Prolog, two points must be explained in the case of *Quixote*.

The first point is the role of object terms as object identifiers. The value of an attribute of an object must be unique, since the label of the attribute is interpreted as a function.

The second point is the fact that the attributes of an object can be partially specified and they are interpreted as subsumption constraints.

Consider the following database:

Example 1

$$o[l = X]/[l_1 \rightarrow a, l_2 = b] \Leftarrow X \parallel \{X \sqsubseteq c\};; \\ o[l = X]/[l_1 \rightarrow d, l_3 = e] \Leftarrow X \parallel \{X \sqsubseteq f\};; \\ p;;,$$

where both $p \sqsubseteq c$ *and* $p \sqsubseteq f$ *hold.*

In this case, $o[l = p]/[l_1 \rightarrow a, l_2 = b]$ holds by the first rule and $o[l = p]/[l_1 \rightarrow d, l_3 = e]$ holds by the second rule. Thus, by combining these two, the object term $o[l = p]$ gains $[l_1 \rightarrow (a \downarrow d), l_2 = b, l_3 = e]$ as its attribute.

This process is done by *merging* the attributes of the derived subgoals equivalent to each other.

The merging process becomes complicated if we take into account the partiality of the attributes of an object.

Consider the following example:

Example 2

$$o/[l_1 \rightarrow a] \Leftarrow p/[l_2 \rightarrow b];; \\ o/[l_1 \rightarrow c] \Leftarrow p/[l_2 \rightarrow d];; \\ p;;.$$

The subgoal p of the first rule holds with attribute $[l_2 \rightarrow b]$, which is not defined in the database. This is because the fact $p;;$ in the example does not specify the value of its l-attribute. Similarly, the subgoal p of the second rule holds with $[l_2 \rightarrow d]$. If these two attributes are inconsistent, the two rules cannot be applied together, that is, the derivations given by the two rules must not be merged.

Definition 3 (Derivation of a goal)
A derivation of a goal G_0 *by a program is defined as the 5-tuple* (G, R, Θ, HC, BC) *of a sequence* G $(= G_0, G_1, \ldots)$ *of goals, a sequence* R $(= R_1, \ldots)$ *of the renaming variants of the rules, a sequence* Θ $(= \theta_1, \ldots)$ *of most general unifiers*[3]*, the two sets of constraints* HC *and* BC *of all the head constraints and all the body constraints of the rules in* P*, such that each* G_{i+1} *is derived from* G_i *and* R_{i+1} *using* θ_{i+1}*, and* $(HC \cup BC)\Theta$ *is solvable.*

Definition 4 (Assumed constraint set)
The assumed constraint set of a derivation D $(= (G, P, \Theta, HC, BC))$ *is defined as the set of all constraints in* BC *that are not satisfied by* HC *with respect to the substitution* Θ.

The assumed constraint set of a derivation is the set of attributes of objects which are assumed to derive the goal. This is because some attributes of objects in a database are partially defined.

Each derivation has its own *derivation context* defined as the consequence relation ($\vdash_C$) between its assumed constraint set and its head constraints. A derivation context $A \vdash_C B$ of a goal represents that the goal is derived by assuming A, and as a consequence, B holds.

The notion of a *refutation* is defined similarly as usual: a derivation that has the empty goal as the last element in its sequence.

In Example 2, the two refutations of the goal o have the following derivation contexts, :

$$p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq a, \\ p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c.$$

To deal with the merging of attributes discussed above, a goal must be merged into the other refutation of the same goal if the derivation contexts of the two refutations have some relation to each other, that is, if the assumed constraint set of one refutation holds in the assumed constraint set of another refutation. This means that the condition holds in a weaker assumption also holds in a stronger assumption.

For example, in Example 2, if $b \sqsubseteq d$ holds, then the second refutation is merged into the first one. As a consequence, a new refutation is given instead of the first refutation, whose derivation context is as follows:

$$p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq (a \downarrow c).$$

Moreover, if $b \sqsubseteq d$ and $c \sqsubseteq a$, then the context of both refutations becomes:

$$p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c.$$

[3]The most general unifier of two object terms is defined similarly to the usual one, except for the definition of terms.

This means that the first derivation is absorbed to the second with respect to the merge, because $(a \downarrow c) = c$ holds.

After merging all possible pairs of refutations, the notion of an answer to a query is defined as follows:

Definition 5 (Answer)
An answer to the query is defined as a pair of the answer substitution and the derivation context of a refutation.

Thus, the following two answers are given to the query $?-o/[l = X]$ to the database shown in Example 2:

$$(\{\}, p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq a),$$
$$(\{\}, p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c),$$

if no condition is given among a, b, c , and d.

The *QUIXOTE* interpreter returns all answers at once, that is, it employs the top-down breadth-first search strategy.

5 Modules in *QUIXOTE*

In this section, a module concept is introduced into *QUIXOTE*.

5.1 Need for Modules in Deductive System

The goal of knowledge representation is to provide a facility for *reasoning* about a problem by using given knowledge in the way that ordinary people do: we call this everyday-reasoning, or human-reasoning.

Such reasoning systems can be defined as the pair (R, A) of a set of deductive rules and an algorithm for extracting all consequences from the rules.

For simplicity, fix A, and think of R as the knowledge in a reasoning system.

- R is neither consistent nor complete, even though its fragments may be consistent in themselves,
- reasoning is situation-dependent, i.e., some fragment of R is relevant or meaningful in a certain situation,
- reasoning usually requires some assumptions.

One way to deal with such an aspect of reasoning is to associate an index to each literal and each rule in R.

Indexes can be used:

(1) to define a fragment of rules (a chunk of knowledge) which can be used in a certain situation, and

(2) to clarify which assumption (set of rules) is used.

(1) defines our conception of a *module* as a set of rules with the same index. Thus, if we regard an index as the identifier for a context or a situation, the set of rules can be seen as the chunk of knowledge relevant for that context or situation.

As the result of introducing indexes, each literal has come to have the form:

$$m : A$$

where m is an index called *module identifier*, and A is an object term or an attribute term.

Hence, the usual consequence relation between formulas should be replaced by:

$$m_1 : A_1, \ldots, m_i : A_i \vdash m : A$$

Intuitively, this means that A holds in m with reference to parts $m_1, \ldots, m_i$ of the database. In obtaining the answer, the choice of parts of the database can be seen as the assumptions.

In *QUIXOTE*, an object term is used as a module identifier. The use of object terms as module identifiers enables the user to treat modules as objects, and provides meta-like programming facilities.

5.2 Rules with Module Identifiers

Corresponding to the constraint-based form of a rule given in Section 4, a modularized rule has the following form:

$$m_0 :: o_0 \Leftarrow m_1 : o_1, \ldots, m_n : o_n \parallel C$$

where $o_0, o_1, \ldots, o_n$ are object terms, $m_0, m_1, \ldots, m_n$ are module identifiers, and C is a set of constraints[4].

This rule specifies the following two things:

(1) this rule is in (or is accessible from) the module with a module identifier m_0, and

(2) if each subgoal $m_i : o_i$ holds with respect to a variable assignment and constraints C then $m_0 : o_0$ holds.

Generally, the modules and their rules are defined as follows:

$$\{m_1, \cdots, m_n\} :: \{r_1; ; \cdots ; ; r_m\},$$

where $r_1, \cdots, r_m$ are rules. Note that it is possible for modules to be nested.

Thus, it is easy to have a set of rules in a module as the set of all rules with the module identifier of that module.

[4]Precisely, this form represents the rule

$$o_0 \Leftarrow m_1 : o_1, \ldots, m_n : o_n \parallel C$$

with index m_0.

The set of rules in the module with m as its identifier is written as Σ_m[5]. In general, a module identifier may be a parametric object term, that is, an object term with variables in its description. The variables appeared in a rule are interpreted as universally quantified, thus the parametric module identifiers which are equivalent with respect to variable renaming are regarded as the same.

In *QUIXOTE*, it is assumed that each module is consistent. It is an important feature of modules to represent inconsistent knowledge where inconsistency arises from differences in situations or context. For example, consider the situation of John's believing that Mary is 20 years old, when she is actually 21 years old. The following database shows the treatment of such a problem:

$$johns\_belief :: mary/[age = 20];;$$
$$real\_world :: mary/[age = 21];;$$

In this case, the database is consistent as a whole unless the two modules are related to each other.

The following example shows the use of parametric module identifiers to describe so-called *generic* modules. A parametric module identifier can be used to pass *parameters* to the rules in the module.

Example 3 (Generic Module)

$$\begin{array}{l} sorter[cmp = C] :: \{ \\ \quad sort[l = [], sorted = [], cmp = C];; \\ \quad sort[l = [A|X], sorted = Y, cmp = C] \Leftarrow \\ \qquad split[l = [A|X], base = A, cmp = C, l_1 = L_1, l_2 = L_2], \\ \qquad sort[l = L_1, sorted = Y_1, cmp = C], \\ \qquad sort[l = L_2, sorted = Y_2, cmp = C], \\ \qquad list : append[l_1 = Y_1, l_2 = Y_2, l = Y];; \\ \qquad \ldots\};; \\ \quad \vdots \\ less\_than :: \{ \\ \quad compare[arg1 = A, arg2 = B, res = yes] \parallel \\ \qquad \{A < B\};; \\ \quad compare[arg1 = A, arg2 = B, res = no] \parallel \\ \qquad \{B < A\}\};; \end{array}$$

Module $sorter[cmp = C]$ has the definition of a quick-sorting procedure which uses the argument C as the comparator, and module *less_than* has the definition of a comparator, where the relation $<$ is used as the constraint relation for comparing two objects.

In processing the query:

$$\begin{array}{l} ?{-}sorter[cmp = less\_than] : \\ \qquad sort[l = L, sorted = R, cmp = C], \end{array}$$

the module identifier *less_than* is passed to the rules in the sorting module, and used to compare two elements of list L. It is possible to give module identifiers other than lt for using different comparator in the sorting procedure.

The next example shows the treatment of *state transitions* by using modules to represent states.

Example 4 (State Transition)

$$\begin{array}{l} m :: \{ \\ \quad a/[on = nil];; b/[on = a];; \\ \quad c/[on = nil];; d/[on = c]\};; \\ sc[sit = M, op = move[obj = A, fr = B, to = C]] :: \{ \\ \quad C/[on = A] \Leftarrow \\ \qquad M : A/[on = nil], \\ \qquad M : B/[on = A], \\ \qquad M : C/[on = nil];; \\ \quad B/[on = nil] \Leftarrow \\ \qquad M : A/[on = nil], \\ \qquad M : B/[on = A], \\ \qquad M : C/[on = nil];; \\ \quad A/[on = nil] \Leftarrow \\ \qquad M : A/[on = nil], \\ \qquad M : B/[on = A], \\ \qquad M : C/[on = nil]\};; \end{array}$$

In the initial state m, block a is on top of block b, and block c is on top of block d. $move[obj = A, fr = B, to = C]$ represents the operation of moving A from the top of B to the top of C.

Module $sc[sit = M, op = OP]$ defines how the state of M is changed by operation OP. At the same time, the module identifier shows the history of state transitions.

For example, the following answers are obtained:

$$\begin{array}{l} ?{-}sc[sit = m, op = move[obj = a, fr = b, to = c]] : \\ \qquad X/[on = a]. \\ \text{Answer} : X = c. \end{array}$$

In this case, the module that represents the state after an operation is not included in the given program, it is possible to create new modules by adding a program to a query (Section 7) and by issuing a *create_module* command.

Concerning modifications made by the sequence of queries
and *create_module* commands, *QUIXOTE* employs transaction logic with special commands, *begin_transaction*, *end_transaction*, and *abort_transaction*. If some modules are created in one transaction, they are incrementally added to the program unless the transaction ends with *abort_transaction*.

[5]Precisely, Σ_m should be defined as the set of rules that are *properly* in m. Taking rule inheritance into account, the set of rules in a module is the union of the proper set and sets of rules imported from the other modules.

6 Relating Modules

It is important to relate some modules in defining the database and when reasoning.

Two ways of relating modules should be considered, that is, referring to other modules and importing/exporting rules from other modules.

As shown above, a rule of QUIXOTE has a subgoal of the form $m : A$ in its body. This subgoal specifies the external reference to the module with m as its identifier. In such a case, module m can be seen as *encapsulated*, because no rule is imported to it.

6.1 Simple Submodule Relationship

Sometimes, it is useful to define databases by providing a facility to import/export among modules as in typical object-oriented languages.

In QUIXOTE, importing/exporting rules are done by *rule inheritance* defined in terms of the binary relation $\sqsubseteq_S$ over modules called the *submodule* relation. The submodule relation is similar to the subsituation relation in PROSIT[15][6]. Basically, rule inheritance is defined as follows:

Definition 6 (Rule Inheritance)
If $m_1 \sqsupseteq_S m_2$ then module m_1 inherits all the rules of m_2, that is, all the rules in m_2 are exported to m_1.

Under this definition, the set of rules of m_1 is $\Sigma_{m_1} \cup \Sigma_{m_2}$.

The right hand side of $\sqsupseteq_S$ in a submodule definition may be a formula of module identifiers with set-theoretical union, intersection, or difference. For example, if we have

$$m_1 :: \{r_{11}, \cdots, r_{1i}\},$$
$$m_2 :: \{r_{21}, \cdots, r_{2j}\},$$
$$\{m_2, m_3\} :: \{r_{31}, \cdots, r_{3k}\},$$
$$m_1 \sqsupseteq_S m_2 - m_3$$

then m_1 has the set of rules

$$\{r_{11}, \cdots, r_{1i}, r_{21}, \cdots, r_{2j}\}.$$

Taking the rule inheritance into account, a special module identifier *self* is also introduced as in most object-oriented programming languages. For example, consider the following:

$$m_1 :: o \Rightarrow o_1 \parallel C.$$

The subgoal o_1 is interpreted as $self : o_1$. In this context, $self$ is evaluated as m_1. If $m \sqsupseteq_S m_1$, then m has the rule $m :: o \Rightarrow o_1 \parallel C$, and $self$ is evaluated as m in this case.

[6] Considering a module as a class, $m_1 \sqsupseteq_S m_2$ means that m_2 is a super-class of m_1.

6.2 Controlling Rule Inheritance

To treat various rule inheritance phenomena, two orthogonal modes, *local* and *overriding*, are introduced into QUIXOTE. Each rule may have these modes, which control how each rule is inherited according to submodule relations.

If a rule is *local*, then it is not inherited to other modules. An *overriding* rule overrides the other rules inherited from other modules, that is, the inheritance of some rules is canceled.

There are several possibilities on what rules are to be canceled by an overriding rule. Currently, the inheritance of a rule is canceled if its head has object terms with the same principal object and its labels are same as the one of the head of overriding rule. This is similar to the 'retract' predicate of Prolog.

Each rule has an *inheritance mode*. The value of the inheritance mode is (o), (l), or (ol), if explicitly specified. (o) means 'overriding, (l) means 'local', and (ol) means 'local and overriding'. If a rule has no inheritance mode, the rule is regarded as having 'non-local and no-overriding' by default.

Consider the following example.

Example 5 (Exception by Inheritance Mode)

$$bird :: canfly/[pol = yes];;$$
$$penguin :: (ol)\ canfly/[pol = no];;$$
$$super\_penguin :: \{\ldots\};;$$
$$bird \sqsubseteq_S penguin \sqsubseteq_S super\_penguin;;$$

The inheritance of the rule of the module *bird* is canceled in the module *penguin* by its 'overriding' rule, whereas the module *super_penguin* gains $canfly/[pol = yes]$, because the rule in *bird* is inherited to it.

By introducing local and overriding modes for rule inheritance, it is possible to relate subsumption and submodule relations closely as follows:

$$penguin \sqsubseteq bird \supset penguin \sqsupseteq_S bird,$$

where rules in m_1 should be overridden.

6.3 Links between two Modules

Sometimes, a facility for representing changes of state is required as shown in the example in Section 5.2.

The relation between the two states before and after an operation is represented by a special form of object terms. However, simpler and more sophisticated treatment may be required for general treatment of state transitions or changes of states. The problem is how to *relate* modules and objects.

Another kind of relations called *links* are provided as follows:

$$m_1 \xrightarrow{L} m_2,$$
$$o_1 \xrightarrow{L} o_1,$$

where m_1 and m_2 are module identifiers, and o_1 and o_2 are object identifiers. L is called the name of a link relation. Notice that link relations are defined over module identifiers and object terms. The former links are called *module-links* and the latter links are called *object-links*.

The links defined above obeys the following rule:

$$m_1 \xrightarrow{L} m_2 \ \wedge \ o_1 \xrightarrow{L} o_2 \ \wedge \ m_1 :: o_1 \ \supset \ m_2 :: o_2.$$

This rule shows how module-links and object-links collaborate. According to this rule, a pair of a module-link definition and an object-link definition can be transformed as follows:

$$m_2 :: X \Leftarrow m_1 : Y,\ m_1 \xrightarrow{L} m_2,\ Y \xrightarrow{L} X.$$

The following is an example of link usage:

$$m_1[agt = a] \overset{\text{turn-back}}{\longrightarrow} m_2[agt = a]$$
$$to\_the\_right\_of[obj = b] \overset{\text{turn-back}}{\longrightarrow} to\_the\_left\_of[obj = b].$$

This example means that b is to the right of an agent a in a module m_1, while b is to the left of a in m_2 after a' turns back.

By traversing the used links, one can keep track of the stages of reasoning. This feature is especially important in assumption-based reasoning and plan-goal based reasoning.

Most of the links appeared in semantic networks can be represented by labels in an attribute term, while some of the links accompanying inference are represented by the pairs of a module-link and an object-link.

7 Programs and Queries

As mentioned before, a *database* or a *program* is defined as a finite set of rules. More precisely, some additional information is associated with the definition of a database or a program.

A definition of a *Quixote* program concept is defined as a 4-tuple (E, M_H, O_H, R) of the environment part E of the definition of macros and information on program libraries, the module part M_H of the definition of the submodule relation, the object part O_H of the definition of the lattice of basic objects, and a set of rules R[7]. The following is an example of a program definition.

$\&b\_pgm;;$
  $\&b\_env;;\ldots;;\&e\_env;;$
  $\&b\_obj;;$
    $\&subsum;;bird \sqsupseteq penguin,\ldots;;$
  $\&e\_obj;;$
  $\&b\_mod;;$
    $\&submod;;penguin \sqsupseteq_S bird,\ldots;;$
  $\&e\_mod;;$
  $\&b\_rule;;$
    $bird :: canfly/[pol = yes];;$
    $penguin :: color[arg = black\_white];;\ldots;;$
  $\&e\_rule;;$
$\&e\_pgm.$

A query is defined as a pair (A, P) of a set of attribute terms A and a program definition P $(=(E, M_H, O_H, R))$.

The purpose of this query is to find the answer to A in the context of adding P. Thus, a query (A, P) to a program P' $(=(E', M'_H, O'_H, R'))$ is the same as a query $(A, [])$ to a program $(E \cup E', M_H \cup M'_H, O_H \cup O'_H, R \cup R')$.

To deal with the modification of the program, a new transaction begins just before a query is processed and ends just after the process is terminated. *Quixote* transactions can be nested, and the user can specify whether the modifications or updates done in each transaction are valid for successive processes or not.

This feature of adding a program fragment in a query extends the ability of the assumption-based reasoning in *Quixote*, as shown in the following query, to the program above.

$?-super\_penguin : canfly/[pol = X];;$
$\&b\_pgm;;$
  $\&b\_mod;;\&submod;;$
    $penguin > -super\_penguin;;$
  $\&e\_mod;;$
  $\&b\_rule;;$
    $penguin :: (ol)canfly/[pol = no];;$
  $\&e\_rule;;$
$\&e\_pgm.$

8 Related Works

8.1 Objects and Properties

Beginning with Aït-Kaci's work on ψ-terms, there are a number of significant works on the formalization complex terms and feature structures [16, 13, 1, 3, 4]. Formalization of the object terms and attribute terms of *Quixote* is closely related to and influenced by those works, especially the work done by Mukai on CIL [14] and CLP(AFA) [13].

Compared to those works, the unique point of *Quixote* is its treatment of object identity that plays an important role in introducing object-orientedness into definite clause constraint languages.

As for object-orientation, Kifer's F-logic is closely related to *Quixote*, although the treatment of object identity and property inheritance is quite different. In F-logic, object identity is not defined over complex terms

[7] Precisely, M_H contains the definition of module-links, and O_H contains the definition of object-links.

but over normal first-order terms. The approach taken in *Quixote* is more fine-grained than that of F-logic.

8.2 Modules

As module concepts are very important in knowledge representation as well as programming, several related works have been done [9, 10, 11, 15]. First, a brief comparison of the language features of these works is presented.

From the viewpoint of knowledge representation, modularization corresponds to the classification of knowledge. In such sense, the flexibility to relate modules is important. *Quixote* provides a number of ways to do this, for example, by specifying the nesting of modules. *Quixote* supports multiple module nesting by allowing set-theoretical operators to relate modules, which are also used for the exception handling, while other languages do not mention to it.

Quixote also provides a facility for dealing with exceptions on exporting/importing rules by using the combination of modes associated with each rule (*local* and *overriding*). This covers the features described in [9, 10, 11].

Furthermore, as in most object-oriented languages, *Quixote* introduces the special module identifier *self* which can be seen as a meta-level variable and plays an important role in rule inheritance, while other languages do not.

On the contrary, other languages have introduced the notion of side-effects mainly to make computation efficient. This is because the others are essentially designed as programming languages. This feature, including database updates, will be enhanced in the next version of *Quixote*.

Concerning the semantics of modules and reasoning with modularized formulas, Gabbay [6] proposes a proof-theoretic framework for extending normal deductive systems called the Labeled Deductive System (LDS). In LDS, each formula is *labeled*, in the form of $t : A$, where t is a symbol called *label* and A is a logical formula. The consequence relation is replaced by:

$$t_1 : A_1, \ldots, t_n : A_n \vdash s : B.$$

In his *concatenation logic*, the following inference rule is the key to relating labeled formulas:

$$s : a, t : a \supset b \vdash (t + s) : b.$$

This means that b is obtained by using s first and then by using t. The label $(t + s)$ indicates the order of label use. This corresponds to the notion of links in *Quixote*, as explained in Section 6.3.

It is worthwhile investigating the relationship between LDS and *Quixote*, namely, to give a proof theory for *Quixote*. This is work to be done in the future.

9 Concluding Remarks

Version 1.0 of *Quixote*, written in KL1 (designed by ICOT as a parallel language for parallel inference machines PIM), has been completed. It has been used for several application systems, such as legal reasoning systems[19], natural language processing systems[18], and molecular biological databases[17]. Through those experiences, the usefulness of the features of *Quixote* are being examined.

We are now working with the new version of *Quixote* for more efficient representation and processing. In the new version, the following features are introduced:

1) Relation between Subsumption and Submodule
   This feature is discussed briefly at the end of Section 6.2.

2) Updates
   In Sections 5.2 and 6.3, we show a simple example of state transition. However, such problems are closely related to updates of databases or programs. Currently, only facts can be added or deleted. In the next version, the facility for adding or deleting non-unit clauses will be provided. The point is how to deal with those updates in a parallel processing environment without causing semantic problems.

3) Meta-Rule
   Meta-rules are useful both in programming languages and knowledge representation languages. They provide a facility to describe schemata to define generic procedures or knowledge.

   For example, in HiLog[5], the following general transitive closure rule can be written:

   $$tc(R)(X,Y) \text{:-} R(X,Y).$$
   $$tc(R)(X,Y) \text{:-} tc(R)(X,Z), tc(R)(Z,Y).$$

   In *Quixote*, new variables corresponding to the principal objects of object terms would be introduced to support such a function.

Acknowledgement

We would like to express our gratitude to the members of the third laboratory of ICOT, and the members of the *Quixote* project for their discussions and cooperation.

We are grateful to the members of the working groups of ICOT, STS (Situation Theory and Semantics) and NDB (New-generation DataBases) and IDB (Intelligent DataBases), for their stimulative discussions and useful comments.

We also would like to thank Dr. Kazuhiro Fuchi, Dr. Koichi Furukawa, and Dr. Shunichi Uchida of ICOT for their continuous encouragement.

References

[1] S. Abiteboul and S. Grumbach, "COL: A Logic-Based language for Complex Objects", *Proc. EDBT*, in *LNCS*, 303, Springer, 1988

[2] P. Aczel, *Non-Well-Founded Set Theory*, CSLI Lecture Notes No. 14, 1988.

[3] F. Bancilhon and S. Khoshahian, "A Calculus for Complex Objects", *Proc. ACM PODS*, 1985

[4] W. Chen and D. S. Warren, "Abductive Reasoning with Structured Data", *Proc. the North American Conference on Logic Programming*, pp.851-867, Cleveland (Oct., 1989).

[5] W. Chen, M. Kifer, and D. S. Warren, "HiLog as a Platform for Database Language", *Proc. the Second International Workshop on Database Programming Language*, Gleneden Beach, Oregon, 1989.

[6] D. Gabbay, "Labeled Deductive Systems, Part 1", CIS-Bericht-90-22, CIS, Universitat Munchen, Feb., 1991.

[7] M. Höhfeld and G. Smolka, "Definite Relations Over Constraint Languages", LILOG report 53, IBM Deutschland, Stuttgart, Germany, Oct., 1988.

[8] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations for Object-Oriented and Frame-Based Languages", Technical Report 90/14 (revised), June, 1990.

[9] D. Miller, "A Theory of Modules for Logic Programming", *The International Symposium on Logic Programming*, 1986.

[10] L. Monterio and A. Porto, "Contextual Logic Programming", *The International Conference on Logic Programming*, 1989.

[11] L. Monterio and A. Porto, "A Transformational View of Inheritance in Programming", *The International Conference on Logic Programming*, 1990.

[12] Y. Morita, H. Haniuda, and K. Yokota, "Object Identity in *Quixote*", *Proc. SIGDBS and SIGAI of IPSJ*, Oct., 1990.

[13] K. Mukai, "CLP(AFA): Coinductive semantics of horn clauses with compact constraints", In J. Barwise, G. Plotkin, and J.M. Gawron, editors, *Situation Theory and Its Applications, volume II.* CSLI Publications, Stanford University, 1991.

[14] K. Mukai, "Constraint Logic Programming and the Unification of Information", *PhD thesis*, Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology, 1991.

[15] H. Nakashima, H. Suzuki, P-K. Halvorsen, S. Peters, "Towards a Computational Interpretation of Situation Theory", *The International Conference on Fifth Generation Computer Systems*, 1988.

[16] G. Smolka, "Feature logic with subsorts", Technical Report LILOG Report 33, IWBS, IBM Deutschland GMBH, W. Germany, 1989.

[17] H. Tanaka, "Protein Function Database as a Deductive and Object-Oriented Database", *The Second International Conference on Database and Expert System Applications*, Berlin, Apr., 1991.

[18] S. Tojo and H. Yasukawa, "Temporal Situations and the Verbalization of Information", *The Third International Workshop on Situation Theory and Applications (STA3)*, Oiso, Nov., 1991.

[19] N. Yamamoto, "TRIAL: a Legal Reasoning System (Extended Abstract)", *France-Japan Joint Workshop*, Renne, France, July, 1991.

[20] H. Yasukawa and K. Yokota, "Labeled Graphs as Semantics of Objects", *Proc. SIGDBS and SIGAI of IPSJ*, Oct., 1990.

[21] K. Yokota and H. Yasukawa, "*Quixote*: an Adventure on the Way to DOOD (Draft)", *Workshop on Object-Oriented Computing'91*, Hakone, Mar., 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Resource Management Mechanism of PIMOS

Hiroshi YASHIRO*†, Tetsuro FUJISE‡, Takashi CHIKAYAMA†,
Masahiro MATSUO‡, Atsushi HORI‡ and Kumiko WADA†

† Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
‡ Mitsubishi Research Institute, Inc.
1-8-1 Shimomeguro, Meguro-ku, Tokyo 153, Japan

Abstract

The parallel inference machine operating system (PIMOS) is an operating system for the parallel inference systems developed in the Japanese Fifth Generation Computer Systems project. PIMOS is written in a concurrent logic language KL1, which adds numerous extensions to its base language, Guarded Horn Clauses, for efficient meta-level execution control of programs. Using such features, PIMOS is designed to be an efficient, robust and flexible operating system. This paper describes the resource management mechanism of PIMOS, which is characterized by its unique communication mechanism and hierarchical management policy.

Hierarchical management of user tasks in a distributed fashion is mandatory in highly parallel systems so that the management overhead of the operating system can also be distributed to the processors running in parallel. The meta-level execution control structure, called shoen, is provided by the KL1 language and is used for providing such hierarchical management in a natural fashion.

In concurrent logic languages, message streams implemented by shared logical variables are frequently utilized the media of interprocess communication. PIMOS, based on this programming style, provides multiplexed streams with flexible control for communication between user programs and the operating system.

1 Introduction

In the Fifth Generation Computer Systems project of Japan, the parallel inference machines, PIMs, have been developed to provide the computational power required for high performance knowledge information systems [Goto *et al.* 1988, Taki 1992].

The parallel inference machine operating system, PIMOS [Chikayama *et al.* 1988], was designed to control highly parallel programs efficiently on PIMs and provide a comfortable software development environment for concurrent logic language KL1.

PIMOS was first developed on an experimental model of parallel inference machine, called Multi-PSI [Nakajima *et al.* 1989], consisting of up to 64 processing elements connected via a two-dimensional mesh network. The system was first developed in 1988 and has been used since then to research and develop various experimental parallel application software. Later, the system was ported to several models of parallel inference machines with considerable improvements in various aspects.

*EMAIL : yashiro@icot.or.jp

1.1 Shoen Mechanism

The language in which PIMOS and all the application programs are written is called KL1. KL1 is a concurrent logic language based on Guarded Horn Clauses [Ueda 1986] with subsetting for efficient execution and extensions for making it possible to describe the full operating system in it.

The greatest benefit of using a concurrent logic language in writing parallel systems is the implicit concurrency and data-flow synchronization features. With these features, one of the most difficult parts of parallel programming, synchronization, becomes automatic, making software development much easier than in conventional programming languages with explicit synchronization.

An important addition by the KL1 language to regular concurrent logic languages is its meta-level execution control construct named shoen. Shoen enables the encapsulation of exceptional events and the description of explicit execution control over a group of parallel computational activities. The execution unit of KL1 programs is a preposition called a goal, which will eventually be proven by the axiom set given as the program. This proof process is the execution process of the programs, as it is with any other logic programming languages. As the proof process can proceed concurrently for each goal, the goals are fine-grained parallel processes.

As no backtracking feature is provided in concurrent logic languages, all the goals in the system form one logical conjunction. Thus, if no structuring mechanism is available, failure in a user's goal means failure of the whole system. The shoen mechanism provides a way of grouping goals, isolating such failure to a particular

group of goals. Such a group is called a shoen.[1]

A shoen can be initiated by invoking the following primitive.

```
execute(Code, Argv, MinPrio, MaxPrio,
  ExcepMask, Control, ^Report)
```

The arguments `Code` and `Argv` represent the code and arguments of the initial goal of the shoen. This goal is reduced to simpler goals during the execution (or proof) process, and all such descendant goals will belong to this shoen.

A shoen has a pair of streams named the *control stream* and the *report stream*, which are represented here by the two arguments `Control` and `Report` respectively. The control stream is used to send commands to control the gross execution of the goals belonging to the shoen, such as starting, stopping, resuming or aborting them as a group. Exceptional events internal to the shoen, such as failure, deadlock, exception such as arithmetical overflow, or termination of computation are reported by the messages received from the report stream (Figure 1).

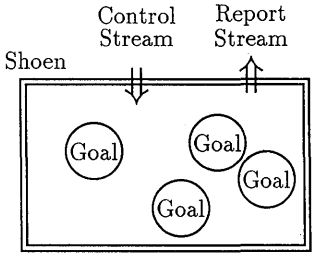


Figure 1: Shoen

The two arguments `MinPrio` and `MaxPrio` specify the priority range of the goals belonging to the shoen. PIMOS does not try to control scheduling of each fine-grained parallel process, but controls them as a group using the control stream and this priority mechanism.

Shoen can be nested by arbitrary levels. Stopping a shoen, for example, will make all the children or grandchildren shoen inside it. The argument `ExcepMask` is used to determine which kinds of exceptional events should be reported to this particular level of the hierarchical structure of the shoen.

PIMOS supervises user programs using this shoen mechanism. The exception reporting mechanism is used to first establish the communication path from the user programs to PIMOS. An exceptional event to be reported can be intentionally generated using the following primitive.

```
raise(Tag, Data, Info)
```

[1]The shoen mechanism is an extension of the meta-call construct of Parlog [Foster 1988] and can be considered to be a language-embedded version of the meta-interpreters seen in systems based on Concurrent Prolog [Shapiro 1984]

The argument `Tag` specifies the kind of event generated by this primitive. This, along with the mask specified when the shoen is created, determines at which level in the shoen hierarchy this event should be processed.

The two arguments `Data` and `Info` are passed as detailed information of the event. The `Data` argument can be any data, instantiated, uninstantiated or partly instantiated, while the `Info` argument has to be instantiated before the event is generated. The above primitive will be suspended until this argument is completely instantiated to be a ground term without any logical variables.

By monitoring the report stream, PIMOS can receive the requests from the user as messages coming from the stream in the following format.

```
exception(Kind, EventInfo, ^NewCode,
  ^NewArgV)
```

The `Kind` argument indicates the kind of exceptional event. In this case, the fact that the event was intentionally generated can be recognized.

The `EventInfo` argument is more detailed information of the event. In the above case, the `Data` and `Info` arguments of the `raise` primitive will be combined together through this argument.

The `NewCode` and `NewArgV` arguments specify an alternative goal to be executed in the object level in place of the goal that generated the event. PIMOS utilizes such a goal for inserting a protection filter, which will be described later.

1.2 Resources

In conventional systems, memory management and process management are two of the most important tasks of the operating system. In the case of PIMOS, as the underlying language implementation of KL1 provides primitives for those fundamental resources, PIMOS do not have to be concerned with such low-level management.

KL1 provides automatic memory management feature including garbage collection, as is the case with Lisp or Prolog. Thus, basic memory management is automatic in the language implementation. KL1 provides implicit concurrency and data-flow synchronization, context switching or scheduling is already supported by the language. Thus, PIMOS does not deal with low-level fine-grained process management, but controls larger-grained groups of processes using the priority system provided by the language.

As memory and process are managed in the KL1 language implementation level, we call them *language-defined resources.* On the other hand, other higher-level resources, such as virtual I/O devices, are more directly controlled by PIMOS. We call them *OS-defined resources.* In what follows, we will concentrate on the management of such OS-defined resources.

2 Communication Mechanism

The basic principles of the communication mechanism are described in this section. This lays the basis for the foundation of the PIMOS resource management mechanism.

2.1 Stream Communication

In a parallel environment, efficient management of various resources becomes much more difficult than in a sequential environment. When data in a particular memory area should not be overwritten while being processed by the operating system, the operating system can simply suspend the execution of user programs in a sequential system. In a highly parallel environment, this will seriously spoil the merit of fine-grained parallelism, as all the user processes sharing the memory space must be stopped irrespective of whether they actually have any possibility of changing the data.

A frequently used programming technique in concurrent logic languages is the object-oriented programming style [Shapiro and Takeuchi 1983]. In this style, a process (actually a goal which becomes perpetual by recursively calling itself) can have internal data which cannot be accessed from outside and shared data containing variables which can be used for interprocess communication. Interprocess communication is effected by gradually instantiating the data shared between processes. Instantiation corresponds to sending data and observing it corresponds to receiving the data. When the shared data is instantiated gradually to a list structure of messages, the structure can be considered to be a communication stream. PIMOS also utilizes this technique for communication between the user programs and the operating system.

For example, reading a character string from the keyboard can be effected by a program shown in Figure 2 (after establishing a communication path by generating an exceptional event as explained in a previous section). The user sends a message `getb/2`, that requests the reading of `N` characters. When PIMOS receives the message, it reads `N` characters from the keyboard to the variable `KBDString` (`readFromKBD/2`). Then, the user receives the `String` instantiated to `KBDString`. As the cdr of the list, `ReqT`, will be a new shared variable after this operation, it can be used for successive such communication.

2.2 Protection Mechanism

In a system based on a concurrent logic language, many of the problems that might arise in a conventional operating system will never be a problem. As the communication path between the user programs and the system programs can be restricted to shared logical variables, there is no way for user programs to overwrite the memory area used by the system programs.

```
?- pimos(Req), user(Req).

user(Req) :-
    true |
        Req = [getb(N,String)|ReqT],
              ......      .

pimos([getb(N,String)|ReqT]):-
    true |
        readFromKbd(N,KBDString),
        KBDString=String,
        pimos(ReqT).
```

Figure 2: An example of interprocess communication between user and PIMOS

With the simple mechanism described above, however, intentional or accidental error in user programs may cause system failure in the following ways.

Multiple Writer Problem When both the system and user programs write different values to the same variable, a unification failure may occur. In a concurrent language like KL1, unifications by PIMOS and the user may be executed concurrently. Thus, this contradiction may cause PIMOS to fail if it tries to instantiate the variable later.

Forsaken Reader Problem The user program may fail to instantiate the arguments of the message sent to PIMOS, in which case PIMOS may wait forever for it to be instantiated.

To solve problems, a filtering process called the protection filter is inserted in the stream between PIMOS and the user program. This filter is inserted in the object-level (within the user's shoen) using the above described `NewCode` and `NewArgV` arguments of the exception reporting message. To solve the forsaken reader problem, the filter will not send a message to PIMOS until its arguments are properly instantiated. To solve the multiple writer problem, the filter will not unify the result from the operating system with the variable supplied by the user until it is properly instantiated by the operating system (Figure 3).

In the actual implementation, such filtering programs are automatically generated from the message protocol definitions.

2.3 Asynchronous Communication

Stream communication is simple, yet powerful enough for simple applications, but it does not provide sufficient flexibility and efficiency at the same time when controlling various I/O devices.

As communication delay is a crucial factor in distributed processing, it is desirable to send messages in a

```
filter([get(C)|User],OS):-
   true |
      OS = [get(C)|OS1],
      wait_and_unify(C1,C),
      filter(User,OS1).
wait_and_unify(OSV,UserV) :-
   wait(OSV) |
      UserV = OSV.
```

Figure 3: An example of the protection filter

pipelined manner for better throughput. To allow this, it is desirable to allow messages to be sent before being sure that they are really needed and to allow them to be canceled if they are found to be unnecessary afterwards. If only one communication stream is available between the operating system and the user, this cancellation is not possible (Figure 4).

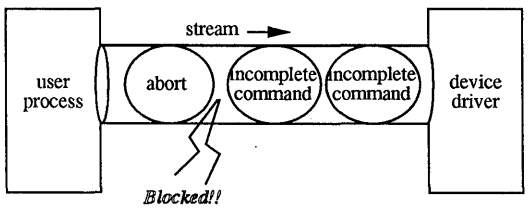


Figure 4: Blocked stream

To solve the problem, PIMOS provides another communication path for emergencies. We call the path *abort line*. This communication path is implemented as a simple shared variable. Instantiation of this variable notifies cancellation of commands already sent to the stream.

Another problem is that, with only one communication stream from the user to the operating system, there is no way for the devices to send asynchronous information to the users. To solve this, besides the above-mentioned two communication paths, a communication path in the reverse direction called the *attention line* is provided (see Figure 5).

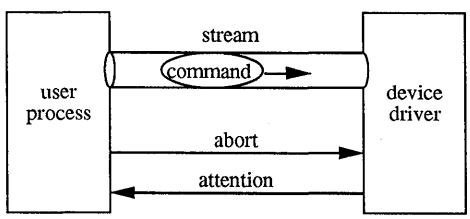


Figure 5: Asynchronous communication with a device

These two "lines" are one-time communication paths in their nature. After they are used, new paths can be established by sending the `reset` message described below through the main communication stream.

2.4 Multiplexing Communication Paths

It is sometimes mandatory to share some (virtual) resources among several processes. A typical example is with the terminal device shared among processes running under a shell. In such cases, only one process should be able to use the device at a time, but quick switching among processes (when a process is suspended by a terminal interrupt, for example) is essential for comfortable operation. On the other hand, the pipelining of I/O request messages is mandatory for better throughput. With only the mechanism of the "abort" and "attention" lines mentioned above, the aborted requests will merely disappear. This does not provide more flexible control, such as suspending a process and resuming it afterwards.

PIMOS provides the following I/O messages to solve the problem.

reset(^Result): The variable `Result` is instantiated to a term `normal(^Abort, Attention, ID)`. The arguments `Abort` and `Attention` correspond to new abort and attention lines. An identifier for a sequence of commands subsequently sent on this stream is returned in the argument `ID`.

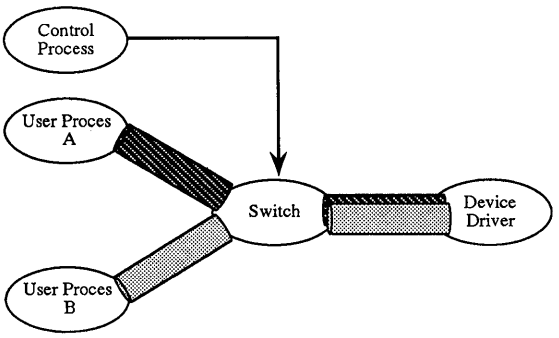
resend(ID, ^Status): When I/O request messages are aborted using the abort line, the device drivers remember the aborted messages associated with the identifier. The `resend` command tells the device driver to retry the aborted messages associated with `ID`.

cancel(ID, ^Status): This `cancel` message tells the device driver to forget about the aborted messages associated with `ID`.

Suppose that a certain device, such as a window device, is shared by two user processes, A and B. Each user process has one communication path to the device. The communication paths connected from the user processes are merged to a "switch" process, which has another communication path connected to a "control" process (Figure 6(a)).

The control process is usually a part of a program such as a command interpreter shell that lets two or more programs share one display window. When a program running under the shell is suspended by an interruption, there may remain I/O messages that have been already sent from the interrupted program to the device driver but have not been processed yet. In such a case, the control process suspends the processing through the abortion line and sends a reset message to the device through

the switch process (Figure 6(b)). The suspended messages are kept in the device driver with ID. If the program resumes communication with the device, the control process commands the switch process to send a resend message with ID as its argument to make it resume the suspended I/O requests.



(a) switch for multiplexing streams

```
              abort                  reset,resend
Process A -------.....................----------
            ID = 1                      ID = 1
               reset, resend    abort
Process B .......------------------.........
                         ID = 2

Example: --- : connected communication path
         ... : disconnected communication path
```

(b) commands between the switch and the device driver

Figure 6: Multiplexing streams

3 Resource Management Mechanism

All the devices provided by PIMOS have the stream interface described above, with attention and abort lines when required. Thus, management of resources in PIMOS is management of these communication paths. This section describes the mechanism of the management by PIMOS.

The following are the keywords to understand the mechanism.

Task: Tasks are the units of management of user programs. A task consists of an arbitrary number of goals (fine-grained processes) corresponding to a shoen in the language level, and forming a hierarchical structure.

General Request Device: The general request device is the top level service agent. This is the stream user programs can obtain directly from PIMOS. Request streams to all other devices are obtained by sending messages to this device.

Standard I/O Device: A task is associated with its standard I/O devices. Standard I/O devices are aliases of some devices they are associated with. The correspondence is specified when the task is generated. The resource sharing mechanism described above is attached to these tasks.

Server: I/O subsystems of PIMOS are actually provided by corresponding tasks called *servers*. They are made relatively independent of the kernel of PIMOS, making the modularity of the system better. The file subsystem is typical of such servers.

3.1 Resource Management Hierarchy

As mentioned above, *tasks* are the unit of management of user programs. All communication paths from user program to PIMOS are associated with certain tasks. Resources obtained by requests through such paths are also associated with the tasks.

Tasks are implemented using the shoen mechanism of KL1. A task is a shoen with its supervisor process inside the PIMOS kernel. The kernel controls the utilization of resources within the task.

Tasks are handled just like ordinary I/O devices. A task handler is a device handler whose corresponding device happens to be a shoen. Tasks are unique in that they may have children resources. As its consequence, a task can have tasks as its children resources forming a nested structure. Corresponding to this, task handlers and other resource controlling processes inside PIMOS also form a hierarchical structure, called the *resource tree*. This resource tree is the kernel of resource management by PIMOS.

One layer of the resource tree is represented by the task handler and *device monitors* corresponding to its children resources connected by streams in a loop structure (Figure 7). Device monitor processes are common with all kinds of devices. Associated with each device monitor is a *device handler*, which depends on the category of the device. Device monitors and device handlers are dynamically created when a new virtual device is created and inserted in the loop structure.

The device handlers can be classified as follows.

Task Handler: A task handler corresponds to a shoen. As described above, usual shoens whose control and report streams are directly connected to their creator. Those streams of shoens corresponding to a task are connected to the task handler. The creator of the task (user programs) can only control and observe the behavior of tasks indirectly through requests to PIMOS.

General Request Handler: General request devices are the primary devices provided by PIMOS. Through them, information on the task itself is ob-

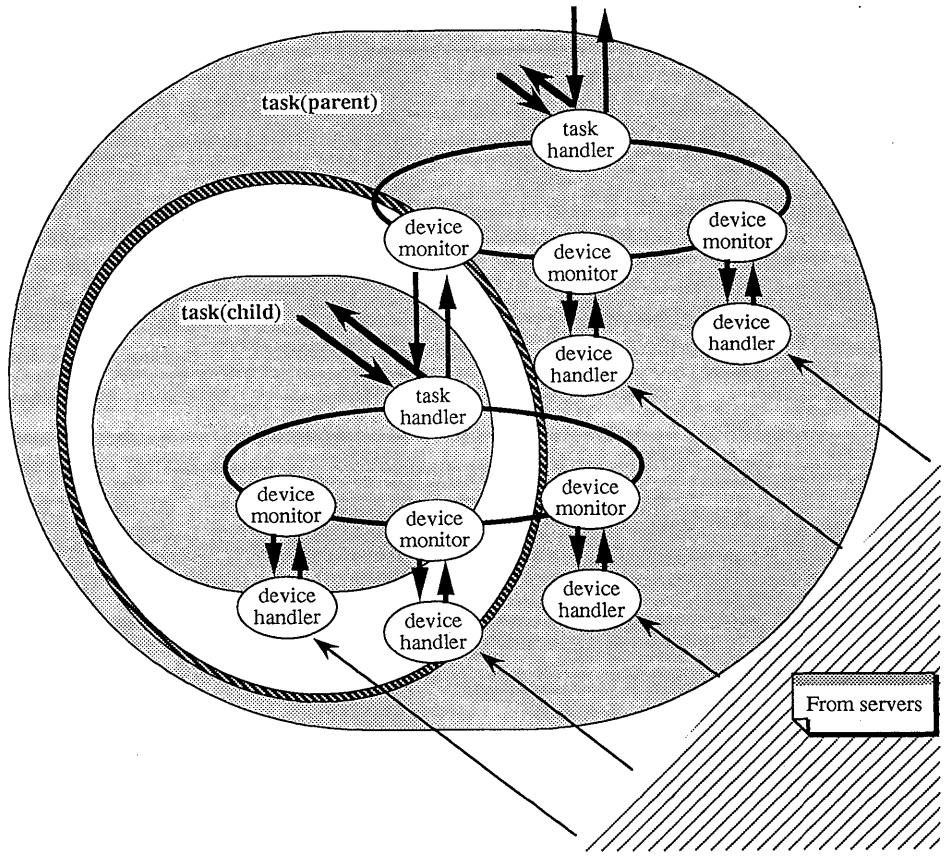


Figure 7: Resource tree

tained and various other devices (including children tasks) can be created.

Standard I/O Handler: Standard I/O devices are aliases corresponding to some other device. They provide the resource sharing mechanism described above.

Server Device Handler: Server devices are the most common form of virtual devices provided by PIMOS. The device handlers watch the status of the client task and notify its termination to the server task.

3.2 Providing Services

To minimize the "kernel" of PIMOS, the kernel provides its fundamental resource management mechanism only. Other services, such as virtual devices such as files or windows, are provided by tasks called "servers".

Figure 8 shows an overview of the management hierarchy of PIMOS. The basic I/O system (BIOS) provides the low-level I/O, but it does not provide the protection mechanism. To protect the system, basic I/O service is provided only for the kernel. The kernel provides the above-described resource tree, which provides the resource management mechanism for tasks. Tasks here include both user program tasks and server tasks.

As described above, communication between the user programs and PIMOS can be established using the *raise* primitive. However, this mechanism only establishes a path to the kernel (the resource tree) and not to a server task.

The communication path between a client task and a server task can be established as follows (see Figure 9, also).

1. To start the service, servers register their service to the service table kept in the kernel of PIMOS. The table associates service names to a stream to the corresponding server. The code for the stream filter for protecting the server from clients' malfunction is also registered in the table.

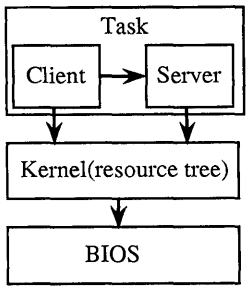


Figure 8: An overview of the management hierarchy

2. The client task establishes a communication path to the PIMOS kernel and requests a service by its name.

3. The kernel searches for the name in the service table, and if a matching service is found, connects the client and the server, inserting a protection filter process inside the client.

Although the above written order is typical, The order of 1 and 2 is not essential. Requests made prior to registration of the service will simply be suspended.

In step 3, PIMOS inserts a device monitor and a device handler corresponding to the server device. The device handler watches for termination of the client task and notifies it to the server (Figure 10) for finalizing the service provided.

This separation of the kernel and the servers in PIMOS allows flexible configuration of the system and assures system robustness. Failures in a server will not be fatal to the system; the services provided by the server will become unavailable, but the kernel of the system not to be affected.

Table 1 lists standard services in the most recent version of PIMOS (Version 3.2). Each of these services is implemented using the above client/server mechanism. Various other servers, such as database servers, can be added easily and canonically to these standard servers.

Table 1: Standard service in PIMOS(Version 3.2)

Name	Service
atom	Database of atom identifiers and their unique printable names.
file	File and directory service.
module	Database of executable program codes.
socket	Internet socket service.
timer	Timer service.
user	Database of user authentication information.
window	Display window service.

3.3 Standard I/O

PIMOS provides a management mechanism for sharing resources, which enables the sharing of resource streams between a parent task and its children tasks (and subsequent children tasks). When a task is generated normally, standard I/O devices of the parent task are inherited to the child task. Multiplexing of the request bstream is implemented as described previously.

Standard I/O devices are not a usual device but a kind of alias of the device it is associated with. Since the protection mechanism of PIMOS, a messages filtering process, has to know the message protocol of the stream, the message protocol for the standard I/O device is restricted to a common subset of I/O device protocols.

3.4 Low Level I/O

In the lowest level, PIMOS supports SCSI (Small Computer Standard Interface) for device control. Each operation to the SCSI bus is provided as a built-in predicate by the KL1 language implementation. For example, a primitive for sending a device command through the SCSI bus is as follows.

```
scsi_command(SCSI, Unit, LUN, Command,
  Length, Direction, Data, DataP,
  ^NewData, ^TransferredLength,
  ^ID, ^Result, ^NewSCSI)
```

The argument `SCSI` should be an object representing the state of the SCSI bus interface device at a certain moment. `NewSCSI`, on the other hand, represents the state of the device *after* sending the command. This is instantiated only after completing the operation and the value will be used in the next operation, which will be suspended until it is instantiated. The proper ordering of operations is thus maintained.

The `Unit` and `LUN` arguments designate a specific device connected to the SCSI bus. Arguments `Command` and `Direction` are used to control communication on the SCSI bus. The argument `ID` is used for command abortion, whose mechanism is similar to one described previously.

Since the KL1 processor needs garbage collection, real-time programming in KL1 is basically impossible. On the other hand, physical operations on SCSI require real-time response. The above primitive only reserves the operation and actual operation will be done eventually, with lower level real-time routines. Explicit buffers are used to synchronize the activities of their lower level routines with KL1 programs. Other arguments, `Data`, `DataP`, `NewData`, `TransferredLength` are used to specify such buffers.

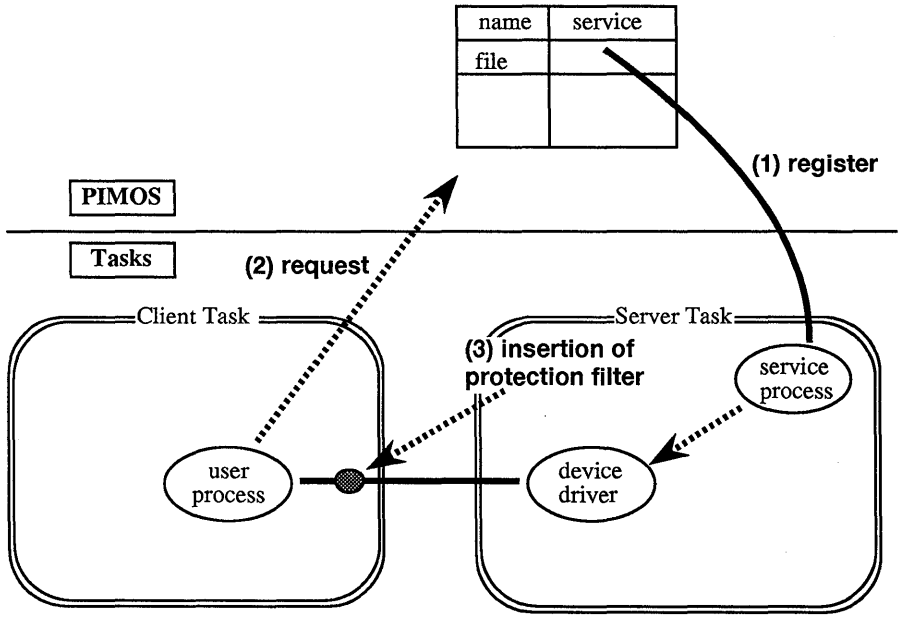


Figure 9: Communication between client and server(1)

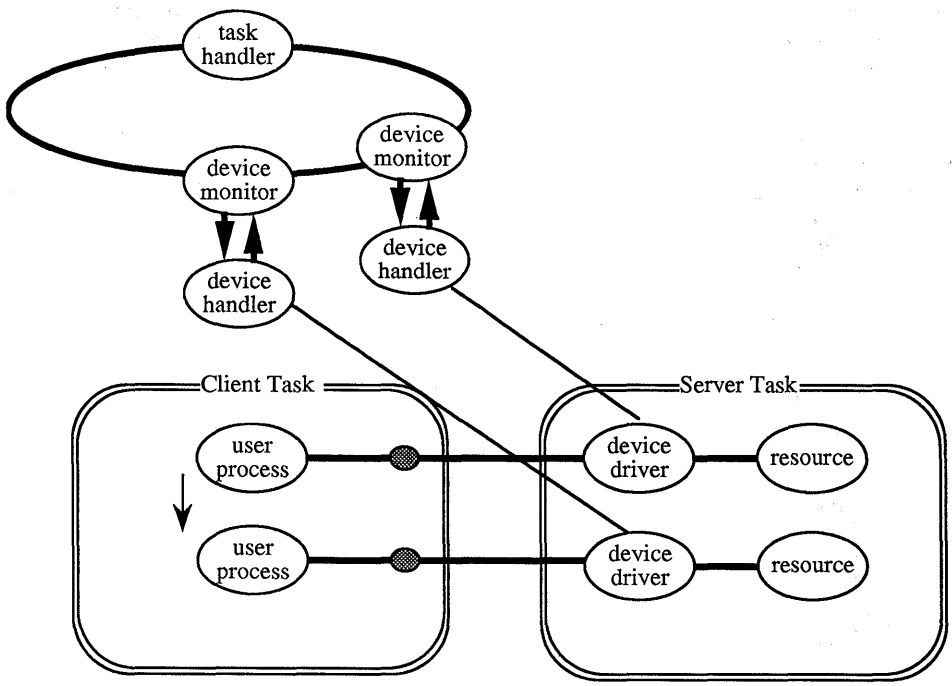


Figure 10: Communication between client and server(2)

3.5 Virtual Machine

As all the communication between the user programs and PIMOS is initiated through the control and report streams of the shoen which implements the user task, a user program can emulate PIMOS and make application programs run under its supervision. This is useful for debugging application programs.

The same technique can also be used to debug PIMOS itself by writing a BIOS emulator, as all the other parts of PIMOS communicate with BIOS through paths established using the shoen mechanism. Figure 11 depicts an actual implementation of a virtual machine on PIMOS. As the virtual machine is a usual task in PIMOS, the protection mechanism of PIMOS prevents failures in the version of PIMOS being debugged on the virtual machine from being propagated to the real PIMOS. This facility has been conveniently used in debugging the kernel of PIMOS.

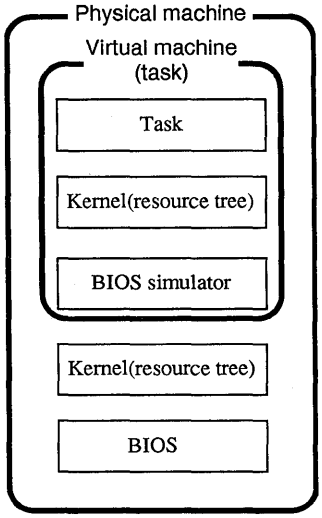


Figure 11: Virtual machine on PIMOS

4 Conclusion

The resource management scheme used in PIMOS based on the concurrent logic language KL1 is described. It depends heavily on the meta-level control mechanism called shoen provided by the language for efficient hierarchical resource management.

PIMOS itself has a hierarchical structure, consisting of a kernel and server tasks. This structure enables a flexible system configuration and reinforces the robustness of the system.

The system consisting of parallel inference machines (Multi-PSI and recently PIM) and earlier versions of PIMOS has been heavily used in research and development of experimental parallel application software for about three and a half years already, proving the feasibility and practicality of implementing an operating system in concurrent logic languages.

Acknowledgement

Many of researchers of ICOT and other related research groups. Too numerous to be listed here, participated in the design and implementation of the operating system itself and development tools. We would also like to express our thanks to Dr. S. Uchida, the manager of the research center of ICOT, and Dr. K. Fuchi, the director of the ICOT research center, for their valuable suggestions and encouragement.

References

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.230–251.

[Foster 1988] I. Foster. Parlog as a Systems Programming Language. *Ph. D. Thesis*, Imperial College, London, 1988.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.208–229.

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989, pp.436–451.

[Shapiro and Takeuchi 1983] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol.1, No.1(1983), pp.25-48.

[Shapiro 1984] E. Shapiro. Systems Programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.

[Taki 1992] K. Taki. Parallel inference machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1992.

[Ueda 1986] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

The Design of the PIMOS File System

Fumihide ITOH Takashi CHIKAYAMA Takeshi MORI Masaki SATO
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Tatsuo KATO Tadashi SATO
Mitsubishi Electric Computer Systems (Tokyo) Corporation
87-1, Kawakami-cho, Totsuka-ku, Yokohama 244, Japan

Abstract

This paper describes the design and implementation of the PIMOS file system. The file system was designed for loosely-coupled multiprocessor systems, where caching is essential for reducing not only disk accesses but also for communication between processors. To provide applications with flexible load distribution, the caching scheme has to support consistency semantics under which modifications of a shared file immediately become visible on other processors. Two different caching schemes, one for data files and the other for directories, have been designed. This is necessary because they have different access patterns. Logging the modifications of directories and other essential information secures the consistency of the file system in case of system failure. Multiple log areas reduce the time required to write logs. Buddy division of blocks enables released blocks to be collected efficiently. Hierarchically organized free block maps control buddy division.

The file system has been implemented on PIM.

1 Introduction

PIMOS [Chikayama *et al.* 1988] has been developed by the Fifth Generation Computer Systems project of Japan as the operating system for PIM [Goto 1989] as a part of the parallel inference system for knowledge information processing. PIMOS has a file system which was designed to realize a robust file system optimized for loosely-coupled multiprocessor systems like PIM. This paper describes the design and implementation of this file system.

The file system for the parallel inference system should provide a bandwidth broad enough to support knowledge information processing application software running on high performance parallel computers. To allow flexible load distribution, the semantics it provides should be location-independent. That is, the contents of files should look the same to the program regardless of the processor it is running on.

File systems on external I/O systems poorly meet requests from multiprocessors, due to limited communication bandwidths. We, thus, constructed an internal file system on disks incorporated into multiprocessor systems. Distributed file systems are similar to our file system in that shared files are accessed from processors connected via a network, with some communication delay. However, the communication bandwidth of the network is much broader in our case. Also, processes using files are normally cooperate rather than compete. These considerations affect the design.

We have clarified functions essential to file systems for loosely-coupled multiprocessor systems, and then considered how to implement them. Although the system is an experimental one, we included the essential features of practical file systems in our design, such as disk access optimization. The system has been implemented as a part of PIMOS, in concurrent logic language KL1 [Ueda and Chikayama 1990].

2 Design Principles

In order to draw parallelism from loosely-coupled multiprocessor systems, centralizing loads to a small number of server processors with disks should be avoided.

The cost of communications between processors is more expensive in loosely-coupled multiprocessor systems than in tightly-coupled ones, and the cost of disk accessing is still more expensive than that of communication. Thus, both disk accessing and interprocessor communication should be reduced. This necessitates distributed caching.

Data cached in memory may be lost upon system failure. For data files, the loss is limited to files being modified at the time of the failure. Loss in a modified directory, however, may cause inconsistency in the file system, such as a deleted, nonexistent file still being registered in a directory. The loss may spread to files under the directory, even though they were not accessed at the time of the failure. Consequently, the file system needs pro-

tection against failure to preserve its consistency.

Disk access optimization is one of the primary features of practical file systems. Most of the overheads in disk accesses are seeks, so a reduction in seek cost, i.e., the number of seeks and per-seek cost, is required.

3 Design Overview

We allowed multiple servers to distribute the loads of file accesses. A caching mechanism was incorporated to reduce disk accesses and communication among processors. A logging mechanism secures the consistency of the file system against system failure. A disk area management scheme similar to that of conventional file systems reduces the seek time for disk accessing. An overview of these features is described in this section.

3.1 Multiple Servers

In order to draw parallelism from multiprocessor systems, load centralization should be avoided. File systems have inherent centralization in that a disk can be accessed only by a processor connected to it. Multiple disks connected to multiple processors with a server running on each relaxes centralization and make the system scalable.

A processor with disks can run a server, but the processor is not dedicated to it. The server processors also operate as clients when their disks are not accessed, providing better utilization of computational resources on multiprocessor systems.

3.2 Caching Mechanism

In order to reduce disk accessing and interprocessor communication, data files and directories are cached onto all processors that access them.

3.2.1 Caching of Data Files

Consistency semantics for caches of the same data on different processors has been realized. The execution of an application program on a multiprocessor system is distributed among processors. The strategies of distribution are diverse and depend upon the application. In order to distribute computation flexibly, file access result must be identical no matter which processor accesses the file. In other words, modification by another processor has to be visible immediately.

This kind of consistency semantics is called Unix semantics [Levy and Silberschatz 1989] in distributed file systems. It was originally introduced to maintain software compatibility between distributed and conventional uniprocessor Unix systems. For the same reason, Unix semantics are indispensable to a file system for multiprocessor systems.

There is no problem in sharing a file when all the sharers merely read the file. When a file is shared in write mode, the simplest way to support Unix semantics is to omit caching and centralize all accesses to the file on the server. This method is reasonable in the environments where shared files are rarely modified. On multiprocessor systems, where processors solve problems cooperatively, modifying shared files is quite common, since distributing the computational load between processors, including file accessing, is essential for efficient execution.

Consequently, a caching mechanism is designed in which a shared file can be cached even if it can be modified and Unix semantics are preserved.

3.2.2 Caching of Directories

In order to identify a file, the file path name is analyzed using directory information. The caching of directories along with the caching of data files can be used to avoid the centralization of loads to server processors and reduce communication with those processors.

Accessing directories is quite different from accessing to data files. Data files are read and written by users, and the contents of files are no concern of the file system. On the other hand, the contents of directories form a vital part of the file system. Thus, a different caching mechanism for directory information was designed.

3.3 Logging Mechanism

Modifications of directories and other information vital for the file system are immediately logged on disk. Modifications are made to data files much more often, and writing all modifications immediately to a disk decreases performance severely. Instead, we provide a mechanism which explicitly specifies the synchronization of a particular file.

Simply writing to a disk immediately does not assure the consistency of the file system. For example, if the system fails while moving a file from one directory to another, the file may be registered in either both or none of the directories, depending on the internal movement algorithm. This inconsistency can be avoided by two-phase modification. First, any modification is written as a log to an area other than the original. Second, the original is modified when logging is complete.

If the system fails before the completion of the logging, the corresponding modification is canceled. If the system fails after completion of the logging but before the modification of the original, the original is modified using the log in a recovery procedure, validating the corresponding modification. In either case, the consistency of the file system is preserved. The system may fail while a log is being written, leaving an incomplete log. In order to detect this, we introduced a flag to indicate the end of a log that corresponds to an atomic modification

transaction.

The completion of logging can be regarded as completion of the modification. The original may be modified at any time before the log is overwritten. This means that logging does not slow down response time. Rather, it improves response time. For example, when a file is moved, two directories have to be modified. The modification of the two originals may need two seeks. Writing the log needs only one seek. Moreover, we use multiple log areas and write the log to the area closest to the current disk head position to reduce the seek time.

A log contains the disk block image after modification. Because the block corresponding to a more recent modification overrides the older modifications, only the newest constituent must be copied to the original. The more times the same information is modified, the less times the original is modified. Frequent modification of the same information, which is known to be the case in empirical studies [Ousterhout *et al.* 1985], minimizes the throughput decline caused by extra writing for logging.

Each log area is used circularly, overwriting the oldest log with a new log. In order to reduce disk accessing, the modifications of the original should be postponed as long as possible, that is, until immediately before the corresponding log is overwritten. To detect the logical tail of a log area, namely the last complete log, each log block has a number, named a log generation, which counts the incidences of overwriting the log area.

The multiplicity of log areas has caused a new problem to arise: how can the newest block be determined after a system failure. If there is only one log area, the newest log block is the closest one to the logical tail of the log area, and the log blocks are always newer than or as new as the corresponding original block. However log blocks in different areas do not show the order in which they were written. If the newest log block is overwritten after it is copied to the original block, the original block is newer than the remaining log blocks. We have solved this problem by attaching a number, named a block generation, to the log blocks and to the original blocks. The block generation counts incidences of modifying the block.

3.4 Disk Area Management

To reduce the number of seeks, the unit of area allocation to files should be made larger. Larger blocks cause lower storage utilization, as a whole large block must be allocated even for small files. Our solution is to provide two or more sizes of blocks and to allocate smaller blocks to small files.

To reduce the time per seek, a whole disk is divided into cylinder groups, and blocks of one file are allocated in the same cylinder group as much as is possible. The log areas mentioned in the previous subsection are placed in each cylinder group.

These methods are commonly used in conventional file systems. A unique feature of the PIMOS file system is buddy division of a large block into small blocks, which reduces disk block fragmentation.

4 Implementation

4.1 Multiple Servers

The whole file system consists of logical volumes, each of which corresponds to one file system of Unix. A logical volume can occupy the whole or a part of a physical disk volume. The processor connected to the disk becomes the server of files and directories in the logical volume. Logging and disk area management in the volume is also the responsibility of the server.

4.2 Data File Caching Mechanism

4.2.1 Overview

To realize Unix semantics with reasonable efficiency on loosely-coupled multiprocessor systems, we decided to stress the performance of exclusive or read-only cases, and tried to minimize disk accesses and interprocessor communication in such cases.

The unit of caching is a block, which is also the unit of disk I/O. This simplifies management and makes caches on server processors unnecessary. A processor where caches are made is called a client, as in distributed file systems. Each client makes caches from all the servers together and swaps cached blocks by the least recently used (LRU) principle. Unix semantics is safeguarded by modifying the cache after excluding caching on other clients.

The caching mechanism is similar to that for coherent cache memory [Archibald and Baer 1986]. While a coherent memory caching scheme depends on a synchronous bus, our platform, a loosely-coupled multiprocessor system, provides only asynchronous message communication. This means that we must consider message overlaps.

A client classifies each cached block into five permanent states, according to the number of sharers and the necessity of writing back to the disk. In addition, there are three more temporary states. In the temporary states, the client is awaiting a response from the server to its request.

A server does not know the exact state of cached blocks, but only knows which clients are caching the blocks. Requests for data, replies to the requests, and other notifications needed for coherence are always transferred between the server and clients, rather than directly between clients. Cached data itself may be transferred directly between clients.

4.2.2 Cache States

The principle for keeping cache coherence is simple: allowing modification by a client only when the block is cached by no other client. To realize this, "shared" and "exclusive" cache states are defined. Permanent cached block states can be as follows:

Invalid (**I**) means that the client does not have the cache.

Exclusive-clean (**EC**) means that the client and no other clients have the unmodified cache.

Exclusive-modified (**EM**) means that the client and no other clients have the modified cache.

Shared-modified (**SM**) means that the client has the modified cache, and some other clients may or may not have cache for the same block.

Shared-unconcerned (**SU**) means that the client has the cache but does not know whether it was modified, and some other clients may or may not have cache for the same block.

Temporary cached block states can be as follows:

Waiting-data (**WD**) means that the client does not have and is waiting for the data to cache, and that the data can be shared with other clients.

Waiting-exclusive-data (**WED**) means that the client does not have and is waiting for the data to cache, and that the data cannot be shared with other clients, as the client is going to modify it.

Waiting-exclusion (**WE**) means that the client already has the cache and is waiting for the invalidation of caches on all other clients. In other words, the client is waiting to become exclusive.

4.2.3 State Transition by Client Request

A request from a user to a client is either to read or to write some blocks. Another operation needed for a cache block is swap-out, i.e., to write the data back to the disk forcibly by LRU. This request or operation is accepted only in permanent states, and is suspended in temporary states as the client is still processing the previous request.

The state transition for a request to read is shown in Figure 1(a). If the state is **I**, the client requests the data to the server, changes its state to **WD**, and waits. After a while, the server reports the pointer to the data and the state to change to. The pointer points to another client when it already has the data, or to the server when the server read the data from the disk because no clients have the data. The client reads the data, lets the user read it, and changes to **EC**, **SM**, or **SU** according to the report. If the state was originally **EC**, **EM**, **SM**, or **SU**, the client simply lets the user read the available data and stays in the same states.

The state transition for a request to write is shown in Figure 1(b). If the state is **I**, the client requests exclusive data to the server, changes to **WED**, and waits. After a while, the server reports the pointer. The client reads the data, lets the user modify it, and changes to **EM**. If the state was originally **EC** or **EM**, the client lets the user modify the data immediately, and changes to or stays in **EM**. If the state was **SM** or **SU**, the client requests the server to invalidate caches in other clients, changes to **WE**, and waits. Then, if the server reports completion of the invalidation, the client lets the user modify the data and changes to **EM**. Another client may also request the invalidation simultaneously, and its request may reach the server earlier. In this case, the server requests the invalidation of the cache, and the client abandons the cache and changes to **WED**. Eventually, after the server receives the request to invalidate from the client, the pointer to the data is reported.

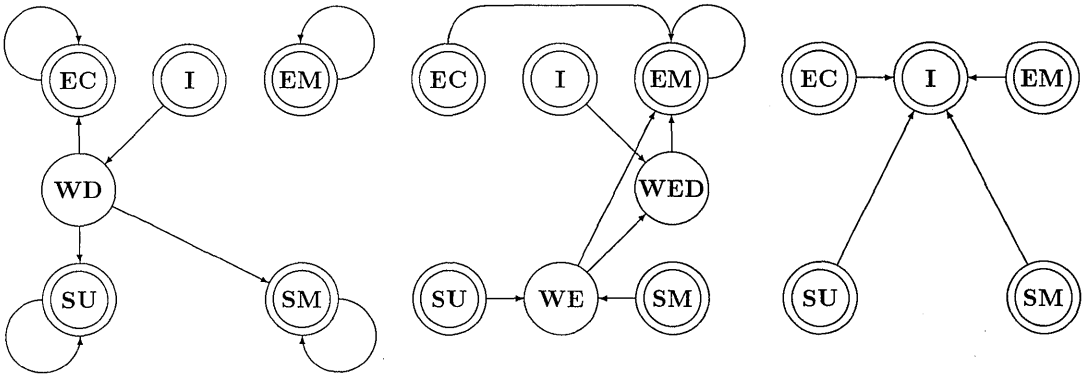
The state transition for swap-out is shown in Figure 1(c). The client reports the swap-out to the server, and changes to **I**. If the state is **EM** or **SM**, the pointer to the data is also reported at the same time. The server reads the data and writes it back to the disk when it cannot make any other client **EM** or **SM**. If the state is **EC** or **SU**, writing the data back to the disk is not required, as the data is either the same as that on the disk or is cached by some other client.

4.2.4 State Transition by Server Request

A request from the server to a client is either to share, to yield, to invalidate or to synchronize the cache. It is accepted not only in permanent states but also in temporary states.

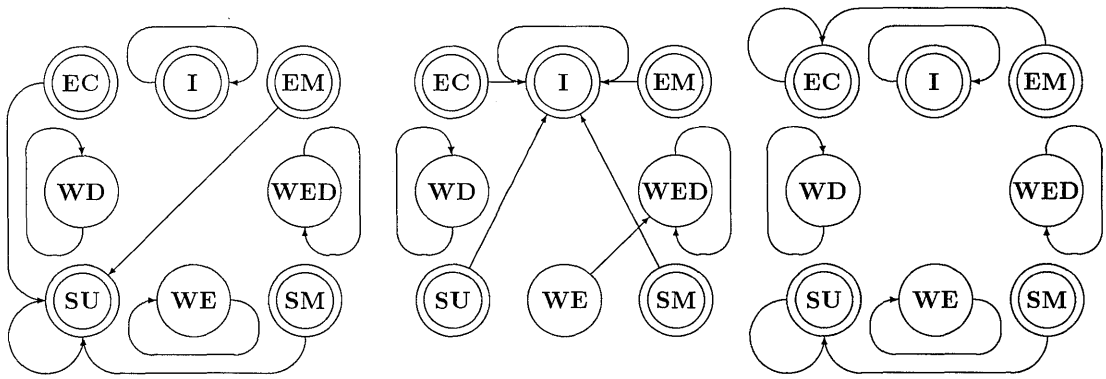
A request to share is caused by a request to read by another client. The state transition for this is shown in Figure 2(a). If the state is **EC** or **SU**, the client reports the pointer to the data and indicates that the requesting client should change to **SU**. In each case, the state of the requested client after replying is **SU**. If the state is **EM** or **SM**, there is a question of which client should take responsibility for writing back the data. In the current design, the requesting client takes it. Consequently, the requested client reports the pointer to the data and indicates that the requesting client should change to **SM**. The requested client becomes **SU**.

A client may receive a request to share in state **I**, if swap-out overlaps with the request. In this case, the server knows of the absence of the data when it receives the swap-out. The client consequently ignores the request. Moreover, the client will possibly receive the request while awaiting data after swapping it out in **WD** or **WED**. The client can also ignore the request. Fur-



(a) Read (b) Write (c) Swap out

Figure 1: State transition diagrams by a request to the client



(a) Share (b) Yield or invalidate (c) Synchronize

Figure 2: State transition diagrams by a request from the server

thermore, the client can receive the request in **WE** if the request to invalidate other caches overlaps with the request. In this case, the client, while waiting in **WE**, reports the pointer to the data and indicates that the requesting client should change to state **SU**. Completion of the invalidation will be reported, after the request of invalidation is received by the server.

The actions of a client for requests to yield and to invalidate are the same, except when the pointer to the data is reported. State transition is shown in Figure 2(b). If the state is **EC**, **EM**, **SM** or **SU**, the client reports the pointer to the data or simply abandons the cache, and changes to **I**. The client can receive the requests in **I**, **WD**, or **WED** and ignore them, for the same reason as when a request to share is received. If the state is **WE**, the client reports the pointer or abandons the cache, and changes to **WED**, as described in the case of a request to write to the client.

On a request to synchronize, the server requests a client to send the data and write it back to the disk. State transition is shown in Figure 2(c). If the state is **EM** or **SM**, the client reports the pointer to the data and changes to **EC** from **EM**, or **SU** from **SM**. If the state is **EC** or **SU**, the client reports that writing back is unnecessary. If the state is **I**, **WD**, or **WED**, the swap-out has overlapped with the request. The client may ignore the request because the server will receive the swap-out message with or without the pointer to the data. If the state is **WE**, the client reports the pointer, while awaiting completion of the invalidation in **WE**.

The temporary states enable message overlaps to be dealt with efficiently.

4.3 Directory Caching Mechanism

Most accesses to directories are to analyze file path names. In order to analyze a file path name on a client, directory information is cached. The unit of caching is

one member of a directory. Each client swaps caches by LRU. The server maintains information on the directories and members that clients cache. The server also caches the disk block images of cached directories, and when a member is added or removed, it modifies the images and writes them to the disk as a log.

When a file path name is analyzed on a client, the members on the path are cached to the client one after another. If the same members appear on subsequent path name analyses, the cached information is used. When a member is added to a directory, the member is added to caches after the addition is logged by the server. These operations require communication only between the client and the server.

When a member is removed, the removal is notified to the server. The server requests the invalidation of the cache to all the clients caching the member. After the server has received acknowledgement of invalidation from all the clients, the server writes a log, thus completing the removal. Although this removal may take time, it is not expected to affect the total throughput of directory caching because frequently updated members of directories are not likely to be cached by clients other than the one that modifies them.

Information about access permission is also necessary for analyzing path names. Therefore, it is cached in the same way as directory information.

4.4 Logging Mechanism

4.4.1 Log Header

In order to manage logging, each block in the logs and the originals has a header consisting of the following items. Except for block generation, the information has no relevance in the original.

Block identifier shows the corresponding original block. It consists of a file identifier and a block offset in the file.

Log generation counts the number of times a log area is used.

Atomic modification end is a flag which shows the last block of a log corresponding to an atomic modification.

Block generation counts the number of times a block has been modified in order to identify the newest block among logs and the original.

Because the size of a header is limited, the maximum size of numbers allowed in the log generation and block generation items are limited. However, as we will discuss, three log generations, at most, can exist in a log area at any one time. This means that the cyclic use of three or slightly more generations is sufficient. Limited numbers

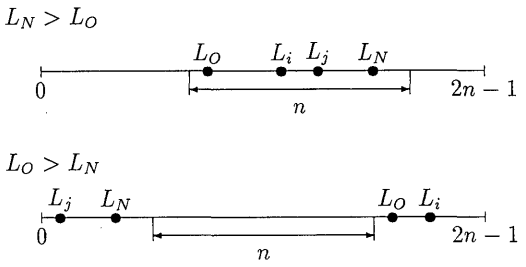


Figure 3: Distribution of block generations

for block generations can also be used cyclically. This assures that the newest block is always spotted in the following way.

Suppose that $2n$ numbers, from 0 to $2n-1$, are used for block generation. Block generation starts from 0, increases by 1 until $2n-1$ is reached, and returns to 0. We have introduced the following control: if the absolute value of the difference between the block generation of the log block to be written next, L_N, and that of the oldest existent block, L_O, is equal to n, the oldest block, whether it is in a log or is the original, is invalidated before the next log is written. Distribution of block generation under this control is shown in Figure 3. As is shown, the invariant condition is that $L_N - L_O < n$ if $L_N > L_O$, and $L_O - L_N > n$ if $L_O > L_N$.

Consequently, after a failure, the newest block is spotted as follows. The distribution of block generations dictates that either all of the generations are in a range narrower than n or that the distribution has a gap wider than n. In the former case, the newest block is the one which has the largest generation. In the latter case, it is the one which has the largest number in the group below the gap.

In practice, the invalidation of the oldest block occurs rarely. Our current implementation allocates 24 bits for block generation. Invalidation occurs only if there are $2^{23} = 8,388,608$ modifications to the same block and, in addition, if the oldest block happens not to have been overwritten by modifications. However, even one modification every ten milliseconds during one whole day barely amounts to $100 \times 60 \times 60 \times 24 = 8,640,000$.

4.4.2 Logging Procedure

While the file system is in operation, logs are written as follows:

1. Create after-modification images of a set of blocks. Set block identifiers and block generations. Line up the blocks and set an atomic modification end flag to the log header in the last block.

2. Choose the log area in the cylinder group where the disk heads currently reside. Set log generations to the log headers. Write the log, the sequence of the blocks, to the log area.

3. Report the completion of logging.

4. Make room in the log area for the subsequent writing. In other words, if the newest blocks are in the part where the next log to the area will be written, copy them to the corresponding original blocks.

5. Invalidate the oldest blocks if necessary, i.e., if there are blocks whose next modifications will require them to be invalidated. Invalidation is performed by setting a null block identifier to the log header.

Making room and invalidating can be done at any time before the next log is written. It should be done immediately after logging to get the best response at the next logging. The size of the room is made to be the maximum size of a log corresponding to an atomic modification, or slightly more.

When a logical volume is dismounted, all the newest blocks are written to the corresponding originals.

The following tables in memory are used to control logging:

Log area table maintains the next log position and the log generation in each log area.

Log record table maintains the block identifier corresponding to each position in the log areas.

Log block table maintains, for every block that has at least one log, the position and the block generation of each log, and the block generation of the original.

4.4.3 Recovery Procedure

After a system failure, the tables for log management are recovered as follows:

1. Find out decreasing points in log generation in each log area.

2. Choose the first of the decreasing points as the tentative logical tail of each log area.

3. Find out the real logical tail of each log area by rejecting the incomplete log from the tentative logical tail.

4. Decide the logical head of each log area and recover the tables from valid log blocks.

Decreasing points in log generation show that the log blocks were logged last before system failure or were being logged at the time of the system failure. There may be more than one decreasing point if an intelligent disk drive changes the order of writing of physical blocks to promote efficiency. In this case, there is also one less increasing point than the number of decreasing points, and the decreasing and increasing points are distributed in the range of one atomic log. Taking into account the circular use of a log area, the log generation of the physical first block is usually one larger than that of the physical last block. If the two generations are equal, the physical tail of a log area is one of the decreasing points in log generation. Examples of the distribution of log generations are shown in Figure 4. There can be one, two, or three log generations in a log area.

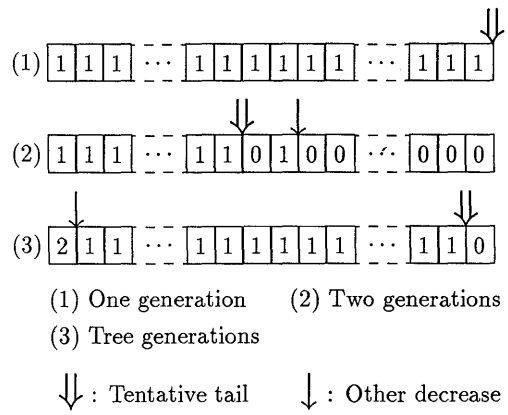


Figure 4: Distribution of log generations

If there is only one decreasing point in log generation, it becomes the tentative logical tail. If there are two or more decreasing points, the first one is selected as the tentative logical tail. The real logical tail is immediately after the last block with an atomic modification end flag before the tentative logical tail. Two tails are identical if the block immediately before the tentative logical tail has the flag.

The logical head is a certain number of blocks away from the real logical tail. The number of blocks corresponds to the room made for the next log writing. Valid log blocks consist of the blocks between the logical head and the real logical tail. After the tables are recovered, the file system can start operation.

4.5 Disk Area Management

To manage the buddy division of large blocks, we use a hierarchy of free block maps in memory as shown in Figure 5. Each free block is registered as free in only one map. We also maintain the number of free blocks registered in each map.

When a free block of a certain size is required and the map of that size has enough free blocks, the map is searched. If it does not have enough free blocks to make

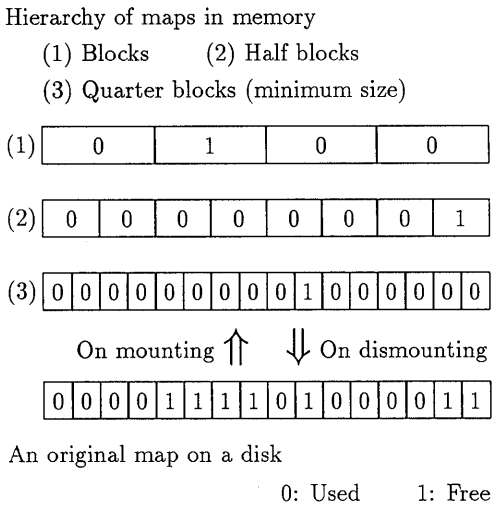


Figure 5: Hierarchy of free block maps

the search efficiently, the map for blocks of twice the size is searched. This continues until the map of the largest block size is reached.

When a block is released and the buddy of the block is free, the two blocks are united and become one free block of twice the size. Otherwise, the released block alone becomes free.

The hierarchy of maps is unfolded from the free block map on a disk whose unit is the smallest block when the logical volume is mounted. It is folded into the original map and saved on the disk when the volume is dismounted.

We use the two-step allocation method common to conventional file systems. In the free block map of the largest block size in memory, only some of the free blocks are registered as free. Another map of the largest block size is made and written to the disk where, in addition to the original used blocks, the free blocks registered as free in memory are registered as used. In this way, the map ensures that the blocks registered as free on it are free, though those registered as used are not necessarily used. Consequently, the file system can start up after a system failure, using the map of the largest block size on the disk, without a time-consuming scavenging operation.

When free blocks in memory become scarce, some are added to the map in memory, and the map entries on the disk corresponding to those blocks are changed to "used". Conversely, when free blocks in memory become surplus, some are removed from the map in memory, and the map entries on the disk corresponding to those blocks are changed to "free". The scarcity and the surplus are judged based on threshold numbers of free blocks in memory.

5 Conclusion

The design and implementation of the PIMOS file system has been described. A multiplicity of servers distributes the file system loads to them and draws out scalability from multiprocessor systems. The caching mechanism, which guarantees Unix semantics, enables applications, including file accessing, to be executed in parallel easily. The logging mechanism secures the consistency of the file system against system failure. The buddy division of free blocks suppresses fragmentation without much overhead.

We are already implementing the file system on PIM. The tuning of parameters and the evaluations of the file system are to be done in the future.

Acknowledgement

We would like to thank Mr. Masakazu Furuichi at Mitsubishi Electric Corporation and Mr. Hiroshi Yashiro at ICOT for their intensive discussions. We would also like to express our thanks to Dr. Shunichi Uchida, the manager of the research department, and Dr. Kazuhiro Fuchi, the director of the research center, both at ICOT, for their suggestions and encouragement.

References

[Archibald and Baer 1986] J. Archibald and J. L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, Vol. 4, No. 4 (1986), pp. 273-298.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 230-251.

[Goto 1989] A. Goto. Research and Development of the Parallel Inference Machine in FGCS Project. In M. Reeve and S. E. Zenith (Eds.), *Parallel Processing and Artificial Intelligence*, Wiley, Chichester, 1989, pp. 65-96.

[Levy and Silberschatz 1989] E. Levy and A. Silberschatz. Distributed File Systems: Concepts and Examples. TR-89-04, Department of Computer Sciences, The University of Texas at Austin, Austin, 1989.

[Ousterhout *et al.* 1985] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer and J. G. Thompson. A Trace-Driven Analysis for the UNIX 4.2 BSD File System. In *Proc. 10th ACM Symposium on Operating Systems Principles*, ACM, New York, 1985, pp. 15-24.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494-500.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems

Seiichi Aikawa Mayumi Kamiko
Hideyuki Kubo Fumiko Matsuzawa
Fujitsu Limited
1015, Kamiodanaka Nakahara-ku,
Kawasaki 211, Japan

Takashi Chikayama
Institute for New Generation
Computer Technology
4-28, Mita 1-chome, Minato-ku,
Tokyo 108, Japan

Abstract

Distributing computational load to many processor is a critical issue for efficient program execution on multiprocessor systems. Naive even distribution of load, however, tends to increase communication overhead considerably, which must also be minimized for efficient execution. It is almost impossible to achieve optimal load distribution automatically. It is especially so on scalable loosely-coupled multiprocessor systems, since the communication cost is relatively high. Finding a good load distribution algorithm is one of the most important research topics for parallel processing.

Tools for evaluating load distribution algorithms are very useful for this kind of research. This paper describes a system called ParaGraph that gathers periodical statistics of the computational and communication load of each processor during program execution, in both the higher level of programming language and lower level of implementation, and presents them graphically to the user.

1 Introduction

In the Japanese Fifth Generation Computer Systems Project, parallel inference systems have been developed for promoting parallel software research and development. The system adopts a concurrent logic programming language KL1 [Ueda 90] as the kernel and consists of a parallel inference machine, PIM [Goto 88] and its operating system, PIMOS [Chikayama 88].

For efficient program execution, the computational load must be appropriately distributed to each processor. On scalable loosely-coupled multiprocessor systems, load balancing and minimization of communication overhead are essential, but become more difficult compared to tightly-coupled systems as communication costs increase. Although many load distribution algorithms have been developed [Furuichi 89, Kimura 89], none have been sufficient to execute every program effectively. Finding a good load distribution algorithm is one of the most important research topics for parallel processing.

Tools for evaluating load distribution algorithms are very useful for this kind of research. The objective of the ParaGraph system is to help programmers design and evaluate load distribution algorithms on loosely-coupled multiprocessor systems. ParaGraph gathers profiling information during program execution on the parallel inference machine, PIM, and displays it graphically.

Many performance displays have been devised for special purpose, processor utilization, communication, and program execution[Malony 90, Heath 91][1]. Profiling information can be viewed as having three axes: what, when, and where. We have designed graphical views based on three axes to display every kind of information with the same form. We also have designed graphical views to be easy to compare the profiling information. This is because bottlenecks are often determined by comparing with the contents of the information relatively in overall execution.

In Section 2, how load distribution can be described in KL1 on PIM are described. Section 3 describes the implementation of the ParaGraph system and graphical representation of program execution, and Section 4 discusses how useful graphical displays are to detect performance bottlenecks with examples of various programs. Section 5 concludes the paper.

2 Load Distribution Algorithms

2.1 Load distribution in KL1

The parallel inference machine runs a concurrent logic programming language called KL1 [Ueda 90, Chikayama 88, Ichiyoshi 89]. A KL1 program consists of a collection of *guarded Horn clauses* of the form:

$$H \ :- \ G_1, \ldots, G_m \mid B_1, \ldots, B_n. \ (m, n \geq 1)$$

where H, G_i, and B_i are atomic formulas. H is called the head, G_i, the guard goals, and B_i the body goals. The guard part consists of the head and the guard goals and the body consists of body goals. They are separated

[1] [Heath 91] describes a tool having the same name as our system, but they are quite different.

by the *commitment* operator(|). A collection of guarded Horn clauses whose heads have the same predicate symbol P and the same arity N, define a procedure P with arity N. This is denoted as P/N.

The guard goals wait for instantiations to variables (synchronization) and test them. When the guard part of one or more clauses succeed, one of those clauses is selected and its body goals are called. These body goals communicate with each other through their common variables. If variables are not ready for testing in the guard part because the value has not been computed yet, testing is suspended.

In addition to the above basic mechanism, there is a mapping facility. The mapping facility includes load distribution specification[2]. The programmer can annotate the program by attaching *pragmas* to the body goals to specify a processor (specified by *Goal*@node(*Proc*)). The programmer must tell the KL1 implementation which goals to execute on which processors.

```
next_queen(N,I,J,B,R,D,BL):- J>0, D=0 |
    BL = {BL0,BL1},
    R = {R0,R1},
    BL0 = [get(Proc)|BL2],
    try_ext(N,I,J,B,R0,D,BL2)@node(Proc),
    next_queen(N,I,~(J-1),B,R1,D,BL1).
```

Figure 1: A sample KL1 program

Figure 1 shows a part of a KL1 program. If the goal next_queen/7 is committed to this clause, its body goals are called. The goal try_ext/7 has a processor specification, and it is to be executed on processor number "Proc". This processor number can be dynamically computed.

2.2 Design Issues

Load balancing derives maximum performance by efficiently utilizing the processing power of the entire system. This is done by partitioning a program into mutually independent or almost independent tasks, and distributing tasks to processors. Many load balancing studies have been devised, but they are tightly coupled to particular applications. Therefore, programmers have to build load distribution algorithms for their own applications.

To distribute the computational load efficiently, the programmer should keep in mind the following points. Since load distribution is implemented by using goals, the programmer should understand the execution behavior of each goal. When goals are executed on a loosely-coupled multiprocessor, the programmer should investigate the load on individual processors and the communication overhead between processors.

For evaluating load distribution algorithms, tools must provide many graphic displays for the programmer to understand the computational and communication load of each processor in both the higher program and lower implementation levels. No single display and no single profiling level can provide the full information needed to detect performance bottlenecks.

[2]The other mapping facility is priority specification to specify what priority the goal should be executed.

3 System Overview

3.1 Gathering Information

To statistically profile large-scale program execution, KL1 implementation provides information gathering facilities, *processor profiling* and *shōen profiling*. KL1 implementation provides these facilities as language primitives, to minimize the undesirable influence to the execution behavior of programs. These facilities have been implemented at the firmware level. The profiling facilities are summarized as follows.

- Processor profiling
  Profiles the low-level behavior of the processor, such as how much CPU time went to the various basic operations required for program execution.

- Shōen profiling
  Profiles the higher-level behavior of the processor, such as how many times each piece of the program was executed.

To minimize the perturbation, the gathered profiling information resides in each processor's local memory during program execution, and after execution, ParaGraph collects and displays this information graphically.

Since profiling information is automatically produced by the KL1 implementation, programmers do not have to modify the application programs.

3.1.1 Processor Profiling

The basic low-level activities can be categorized into *computation*, *communication*, *garbage collection*, and *idling*. Computation means normal program execution such as goal's reductions and suspensions, communication means sending and receiving inter-processor messages, garbage collection means itself, and finally, idling means doing nothing.

The processor profiling facility measures how much time went to each category for each processor. Such information can be periodically gathered to show gradual changes of behavior. The profiling facility can also measure frequencies of sending and receiving various kinds of interprocessor messages [Nakajima 90].

- A *throw_goal* message transfers a KL1 goal with a throw goal pragma to a specified processor.
- A *read* message requests for some value from the remote processor when a clause selection condition requires it.
- An *answer_value* message replies to a read message when the request value becomes available.
- A *unify* message requests body unification (giving a value to a variable).

3.1.2 Shōen Profiling

"Shōen" [Chikayama 88][3] is a mechanism provided in KL1 for grouping goals and controlling their execution in a meta-level. The shōen mechanism can be considered to be an interpreter for the KL1 language. It also provides profiling facility at a higher level than processor profiling. Processor profiling gathers a number of important statistics from many aspects that help analyzing performance bottlenecks, but it provides no information on where in the program is the root of such a behavior.

To correlate execution behavior with a portion of the program, shōen profiling measures how many times goals associated with each predicate are reduced or suspended (due to unavailability of data required for reduction). Transition of behavior can be observed by periodically gathering the information.

3.2 Graphic Displays

The profiling information can be viewed as having three axes: what, when, and where. In sequential execution, "where" is a constant and the "when" aspect is not important, since the execution order is strictly designated. Therefore, simple tools like gprof provided with UNIX[4] suffice. However, all three axes are important when parallel execution is concerned.

If such massive information is not presented carefully, the user might be more confused than informed. Therefore, ParaGraph provides a variety of graphic displays. We named each representation using the terms "What," "When," and "Where." The term "What" is the visualization target corresponding to the type of profiling information such as low-level processor behavior, higher-level processor behavior, and interprocessor message frequencies. The term "When" indicates time expressed by an integer that is a cycle number. The term "Where" indicates the processor number and is expressed by an integer.

Figure 2 shows the graphic displays of ParaGraph. These displays are execution behavior of all solution search program of N queen problem.

[3]The word "shōen" is a Japanese word that means "manor".

[4]UNIX is a trademark of AT&T Bell Laboratories

Every type of profiling information can be easily displayed with the views described below with a menu-oriented user interface such as the bottom-right window in Figure 2. If the window size is too small to display everything in detail, coarser display aggregating several cycles or several processors together is possible to see the overall behavior at a glance. Scrolling on the vertical and horizontal directions are also possible if details are to be examined. It is also possible to display only selected "What" items.

3.2.1 A What×When View

There are two kinds of views in terms of "What" and "When" items. One is a What×When view which shows the behavior of each "What" item during execution. A graph is displayed of a "What" item in order of the total volume. The x axis is the cycle numbers, and the y axis is the rate of processor utilization, the number of messages, and the number of reductions or suspensions corresponding to the type of profiling information. Since every graph is drawn with the same scale on the vertical axis, it is easy to compare with "What" items.

The other is an overall What×When view which shows the behavior of all "What" items during execution. Each "What" item is stacked in the same graph and displayed as a line. The y axis represents the average rate of processor utilization, the total number of messages, and the total number of reductions and suspensions corresponding to the type of profiling information.

These views are helpful for example, if a program has sequential bottlenecks such as tight synchronization. In this case, the number of goal reductions will be down at some portion during program execution. Such a problem will be detected easily by observing program execution.

The top-left window in Figure 2 shows received message frequencies on all processors with What×When view. In this window, four kinds of received message frequencies are displayed on each graph. These messages are displayed in order of the total number of received messages. The other messages are displayed by scrolling vertically.

From this, we know that each received message frequency on all processors is less than 6,500 times/an interval (an interval is 2 second). As this program is divided almost mutually independent subtasks, communication message frequency is very low.

3.2.2 A When×Where View

A When×Where view shows the behaviors of all "What" items on each processor. Each processor is displayed with various color patterns that indicate volume. The relationship between color patterns and volume are shown in the bottom right corner. The darker the pattern, the busier the processor. Volume means the rate of processor utilization, the number of messages, and the number of

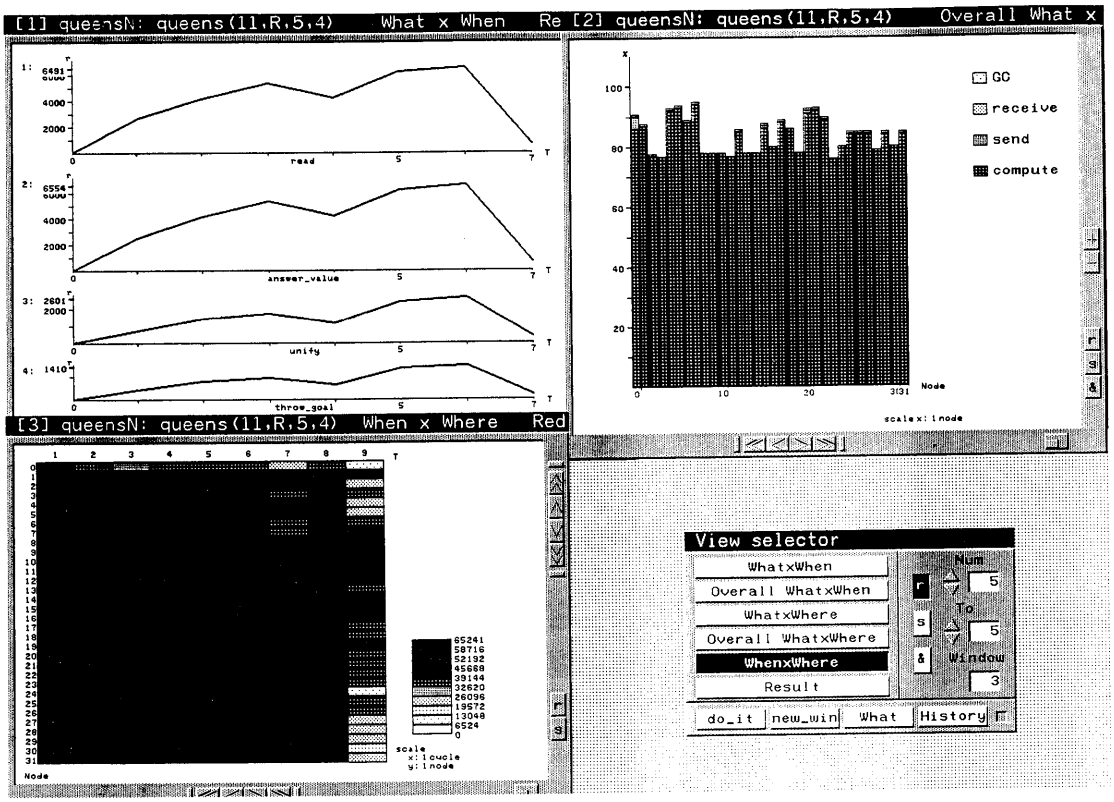


Figure 2: Sample graphic displays: a What×When view (top-left window), an overall What×Where view (top-right window), and a When×Where view (bottom-left window) and a menu-oriented user interface (bottom-left window)

reductions or suspensions that correspond to the type of profiling information. It's also possible to display only selected "What" items instead of all of them.

The bottom-left window in Figure 2 is a When×Where view. The x axis is the cycle number, and the y axis is the processor number. This view displays the execution behavior of all goals on a 32-processor machine. The color patterns indicate the number of reductions. The relationship between the number of reductions and color pattern is displayed on the bottom right corner.

From this, we know that the work load on each processor was well balanced, and this program was executed about 50,000 reductions/an interval on each processor at each moment in time.

3.2.3 A What×Where View

There are two kinds of views in terms of "What" and "Where" items. One is a What×Where view which shows the load balance of each "What" item on each processor. A bar chart is displayed of a "What" item in order of total volume. The x axis represents the processor numbers, the y axis represents the rate of processor utilization, the number of messages, and the number of reductions or suspensions that correspond to the type of the profiling information. All bar charts are drawn with the same scale on the vertical axis, so it is easy to compare with the volume of each "What" item.

The other is an overall What×Where view which shows the load balances of all "What" items on each processor. Each "What" item is stacked in the same bar chart and displayed by a certain color pattern. The y axis represents the average rate of processor utilization, the total number of messages, and the number of total reductions or suspensions that correspond to the type of profiling information. The relationship between each category and color pattern was displayed on the top-right corner.

The top-right window in Figure 2 shows the low-level behavior of the processor with an overall What×Where view. In this window, each categories of low-level behavior is displayed with several color pattern.

From this, the average of computation took more than 80% of total execution time, and the average of commu-

nication on each processor was less than 5%. Thus, this view shows most of the processors run fully, and this example program was executed very efficiently on each processor.

4 Examples

This section discusses which views to use to view various performance bottlenecks. For efficient program execution on multiprocessor systems, the following phases are usually repeated until a solution is reached: 1) a program is partitioned into subtasks, 2) the subtask is mapped to each processor dynamically, and 3) each processor runs subtasks while communicating with each other.

Various problems are often encountered when executing a program on multiprocessor systems. We will show how graphic displays in both the higher program and lower implementation levels are helpful with performance problems.

4.1 Uneven Partitioning

When the granularity between subtasks is very different, it is useful to observe the low-level processor behavior with a When×Where view and the higher-level processor behavior with a What×Where view. From the When×Where view, we will find which processors run fully and which are idle. From the What×Where view, we will determine which goals caused the load imbalances.

The left window in Figure 3 shows the low-level behaviors on each processor with a When×Where view, while the right window in Figure 3 shows the higher-level behaviors of the same processors with a What×Where view on a 16-processor machine. An example program is a logic design expert system which generates a circuit based on a behavior specification. The strategy of parallel execution is that first, the system divides a behavior specification into sub-specifications, next designs subcircuits based on the sub-specifications on each processor, and finally gathers partial results together and combines them.

The When×Where view suggests that processors around No. 11 run fully, but most of the other processors were idle. The What×Where indicates the top six goals were mainly executed on processor No. 11.

From this, we know that very complicated tasks are allocated to processor No. 11, that is, uneven partitioning of behavior specification must cause a bottleneck in performance.

4.2 Load Imbalance

If a mapping algorithm has problems such as allocating subtasks to the same processor, it is useful to observe low-level behavior of the processor with a When×Where view and higher-level behavior with a What×Where view. From the When×Where view, we see which processors are running fully and which are idle, and from the What×Where view, we see the load balance of each goal. Using both views, we can determine how to distribute the goals that are imbalanced to each processor.

The bottom-left window of Figure 4 shows low-level behavior of the processor with a When×Where view, the top-left window and the top-right window show the higher-level behavior of the processor with an overall What×Where view, a What×Where view respectively.

An example program is a part of the theorem prover which evaluates whether an input formula is a tautology. The strategy consists of 2 steps: 1) convert an input formula to clause form (i.e, conjunctive normal form), 2) evaluate its clause form and determine whether it is a tautology.

The step 1 is executed in parallel as follows. First, main task partitions an input formula into subformulas. Second, it generates subtasks to convert subclause forms, and finally, distributes subtasks to many processors dynamically. These steps are repeated recursively until subformulas are converted to subclause forms. The step 2 is executed in sequential on processor No. 0.

The When×Where view of the bottom-left window in Figure 4 suggests that only certain processors (processor No. 6-15 and No. 23-31) run fully and that the others were mostly idle. The overall When×Where view of the top-left window also suggests that most of the goals were executed on certain processors and the number of reduction of top five goals were higher than the other goals.

We can check the load of each goal on each processor from the What×Where view of the top-right window in Fugure 4. These goals were executed on certain processors and were the cause of the load imbalances. From this, we have to change its mapping algorithm to be flatten the shape, to use all processors efficiently.

4.3 Large Communication Overhead

When subtasks are not mutually independent and must communicate with each other closely, the program is less efficient because of communication overhead. In this case, the low-level behavior of the processor with an overall What×Where view and frequencies of sending and receiving messages with a What×Where view are helpful. From the overall What×Where view, we will learn how much time has been consumed on message handling for each processor, while the What×Where view shows us what kind of messages each processor has sent or received.

Figure 5 displays an execution behavior of an improved version of the program described in Section 4.2. The left window shows the load balances of all goals on a 32-processor machine with an overall What×When view.

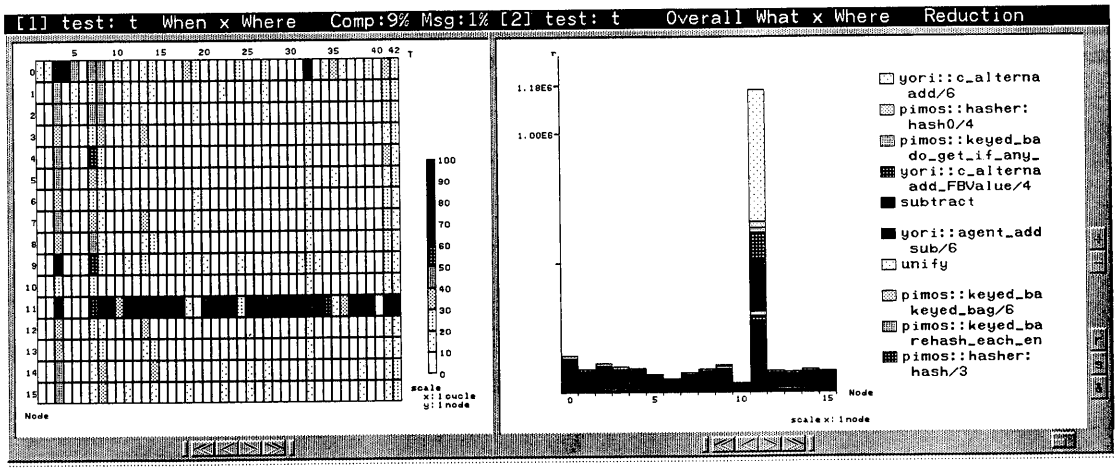


Figure 3: The low-level processor behavior (left window) and execution behavior of goals (right window)

This view shows that the work load on each processor was balanced in overall execution, but was not efficient because of large communication overhead. It will be proved from low-level behavior of the processor with an overall What×Where view shown in the right window.

The right window of Figure 5 suggests the load average on each processor was about 80 - 85%, but the average of computation on each processor was about 20%. Most of the processing power was consumed sending and receiving message handling time more than 60% of total execution time.

Figure 6 shows the same program execution as Figure 5. The left window shows the receiving and sending message handling time rate with What×Where view, the right window shows the frequencies of four received interprocessor messages with a What×When view.

The left window of Figure 6 shows the message handling time on each processor at each moment in time was almost equally, the right window shows that the read message was received about 180,000 times, answer_value message was about 165,000 times, unify message was 100,000 times, and throw_goal message was about 64,000 times per interval on all processors. The tasks generated in this program communicated with each other closely among processors as compared with the result of N queen's message frequencies (see the top-left window of Figure 2).

From this, we know that as work loads are distributed more and more, it becomes easier to balance work loads on each processor, but communication overhead also increases and performance is thus lowered. As a result, we have to redesign or improve how to divide into subtasks. Because the generated subtasks that were not mutually independent, and it caused such a problem we mentioned above.

5 Conclusion

We developed the ParaGraph system on parallel inference machines to provide graphic displays of processor utilization, interprocessor communication, and execution behavior of parallel programs. Experiments with various programs have indicated that graphic displays are helpful in dividing work loads evenly and determining where the bottlenecks are on multiprocessor systems.

We released a version last year as a tuning tool of PIMOS, but have experienced some problems. In the future, we will improve the system considering the following points.

First, real-time performance visualization tools are needed. Although displaying execution behavior in real-time perturbs the program being monitored, it is useful not only in early tuning but also in debugging such as detecting deadlock status and infinite loops. To develop such a tool, low overhead instrumentation techniques and new displays that programmers would not be pressed to understand appearing in real-time must be devised.

Second, tools which can visualize the portion of the performance bottlenecks directly are needed. Massively parallel machines that have thousands of processors and programs for long runs produce a large amount of profiling information, but it is difficult to process or display for simple expansion of our system because of a vast quantity of information. To solve such problems, analysis techniques indicating bottlenecks directly will be needed. We will study automatic analysis techniques and graphical displays of its result (we call this *bottleneck visualization*). One such approach is critical path analysis, which identifies the path through the program that consumed the most time [Miller 90].

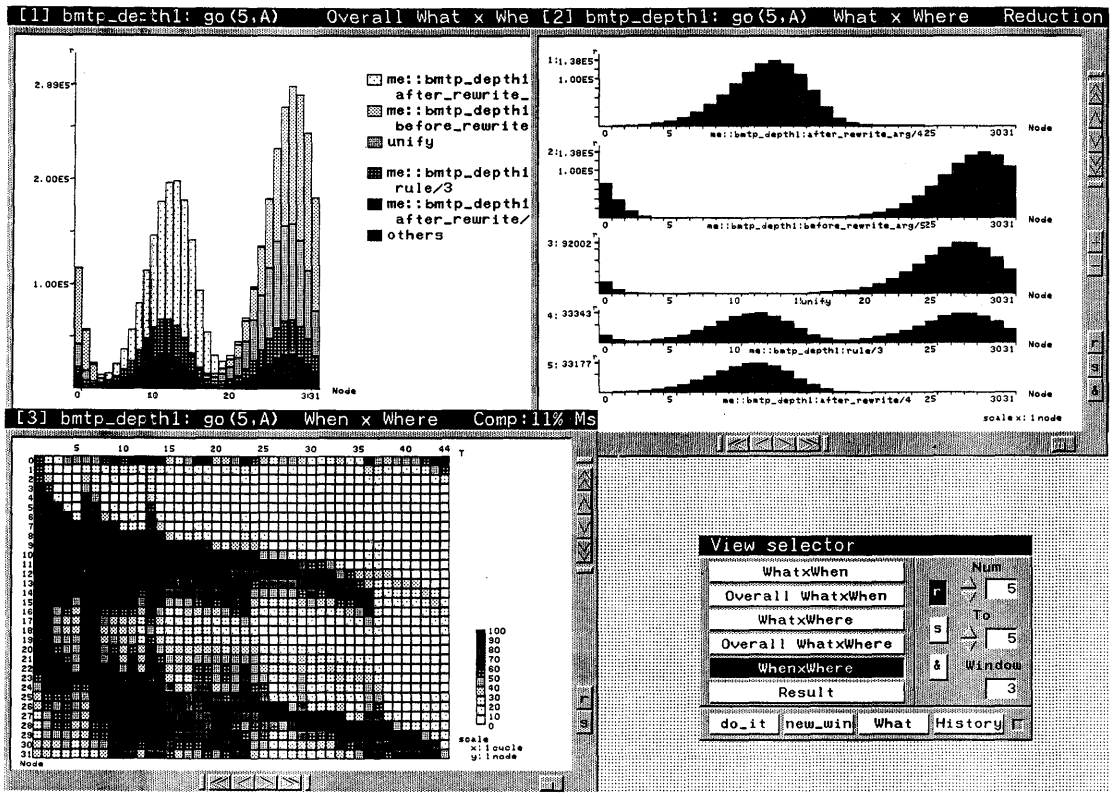


Figure 4: Low-level processor behavior (bottom-left window), the load balances of all goals (top-left window), and the load of each goal (top-right)

6 Acknowledgments

We thank K. Nakao and T. Kakuta who helped us to develop this tool, and all the researchers of ICOT and other companies who tested our tool.

References

[Ueda 90] K. Ueda and T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," The Computer Journal, December 1990.

[Goto 88] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto, "Overview of the Parallel Inference Machine (PIM) architecture," In Proceedings of the International Conference on Fifth Generation Computer Systems, pages 208-229, 1988.

[Chikayama 88] T. Chikayama, H. Sato, and T. Miyazaki, "Overview of the Parallel Inference Machine Operating System (PIMOS)," In Proceedings of the International Conference on Fifth Generation Computer Systems, pages 230-251, 1988.

[Furuichi 89] M. Furuichi, K. Taki, N. Ichiyoshi, "A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Program on the Multi-PSI," ICOT TR-526, 1989.

[Kimura 89] K. Kimura, and N. Ichiyoshi, "Probabilistic Analysis of the Optimal Efficiency of the Multi-Level Dynamic Load Balancing Scheme," In Proceedings of the Sixth Distributed Memory Computing Conference, 1989.

[Heath 91] M. T. Heath, and J. A. Etheridge, "Visualizing the Performance of Parallel Programs," IEEE Software, pages 29-39, September 1991.

[Malony 90] A. D. Malony, D. A. Reed, D. C. Rudolph, "Integrating Performance Data Collection, Analysis, and Visualization," Addison-Wesley Publishing Company, pages 73-97, 1990.

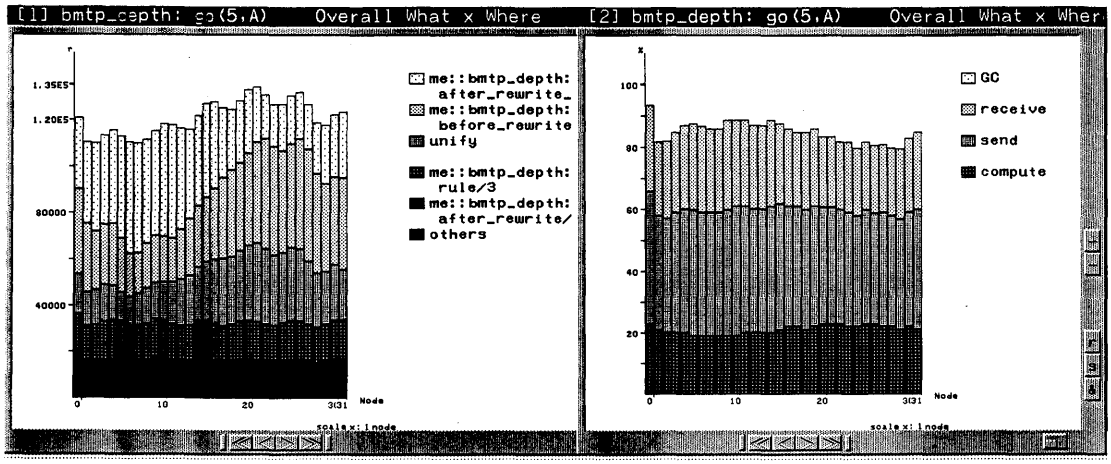


Figure 5: The load balances of goals (left window) and low-level processor behavior (right window)

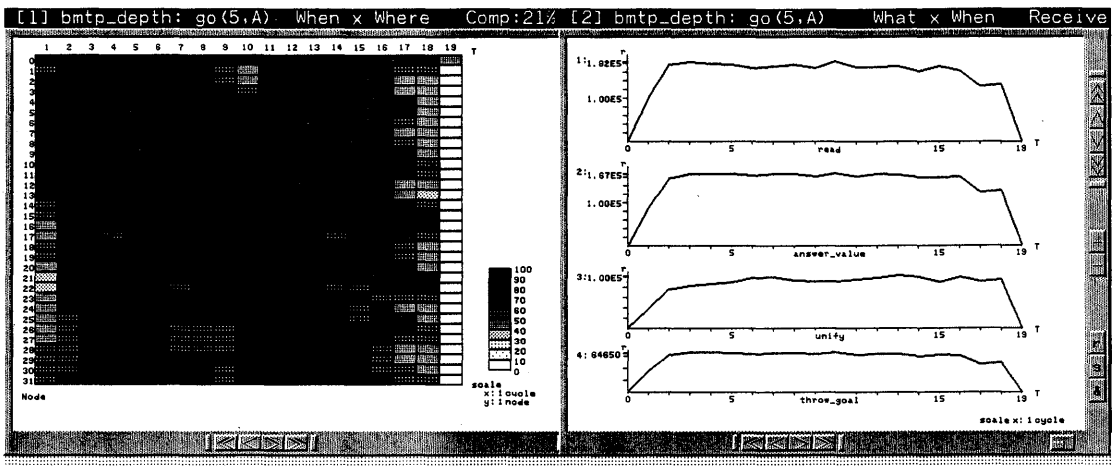


Figure 6: Low-level processor behavior about message handling (left window) and message frequencies (right window)

[Ichiyoshi 89] N. Ichiyoshi, "Research Issues in Parallel Knowledge Information Processing," ICOT TM-0822, November 1989.

[Nakajima 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, T. Chikayama, and H. Nakashima, "Distributed Implementation of KL1 on the Multi-PSI/V2," In Proceedings of the Sixth International Conference on Logic Programming, 1989.

[Nakajima 90] K. Nakajima, and N. Ichiyoshi, "Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI," ICOT TR-531, 1990.

[Miller 90] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Trans. Parallel and Distributed Systems Vol. 1 No. 2, pages 206-217, April 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

PROTEIN SEQUENCE ANALYSIS BY PARALLEL INFERENCE MACHINE

MASATO ISHIKAWA, MASAKI HOSHIDA, MAKOTO HIROSAWA,
TOMOYUKI TOYA, KENTARO ONIZUKA AND KATSUMI NITTA

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
ishikawa@icot.or.jp

Abstract

We have developed a multiple alignment system for protein sequence analysis. The system works on a parallel inference machine PIM. The merits of PIM bring prominent features to the multiple alignment system. The system consists of two major components: a parallel iterative aligner and an intelligent refiner. The aligner uses a parallel iterative search for aligning protein sequences. The search algorithm is the Berger-Munson algorithm with its parallel extension. Our implementation shows that the algorithm extended in parallel can rapidly produce better solutions than the original Berger-Munson algorithm. The refiner uses condition-action rules for refining multiple sequence alignments given by the aligner. The rules help to extract motif patterns from ambiguous alignment patterns.

1 Introduction

Molecular biology and genetic technology have been advancing at an astonishing rate in recent years. Major activities in these fields are closely related to DNA and protein. This is because a set of DNA molecules in a cell contain the genetic information for the complete design of the living organism. This information is embodied as protein to build up the body and to keep its mechanisms alive. Each piece of genetic information, represented by a sequence of nucleic acids, is translated into a sequence of amino acids to form protein. As the method to determine DNA or protein sequences has progressed to its current state, the amount of known sequence data has grown rapidly. For example, *Genbank,* one of the most widely distributed databases, contains information on more than sixty million nucleotides. The growing number of genetic sequences in databases inevitably makes the field of genetic information processing one of the most important application areas for computer science.

The fundamental technique for analyzing genetic sequence data by computer is to examine similarities among sequences. This usually requires large amounts of computation to find the similarities, since there are a lot of sequences in the database to be examined. The computational problem can be partly solved with parallel implementation. There have been some experiments with parallel sequence analysis [Iyengar 1988]. Another approach to the problem is to furnish the analysis program with biological know-how as heuristics. Many consider that logic programming languages are a profitable way of implementing heuristics. Parallel sequence analysis with a logic programming language has been tried [Butler *et al.* 1990].

We have developed a multiple alignment system for protein sequence analysis. The system has been implemented on a parallel inference machine PIM using a parallel logic programming language KL1. The aim of this paper is to show PIM's availability in the field of genetic information processing. The organization of the rest of this paper is as follows. In Section 2, we briefly explain our application problems. We present our multiple sequence alignment system in Section 3. Then, the results of experiments and comparison with other methods are discussed in Section 4. Finally, conclusions are given in Section 5.

2 Protein sequence analysis

As described above, the genetic information, stored in DNA, is translated into sequences of amino acids. A chain of amino acids folds to become protein in water.

The structure of the protein depends on the sequence itself, that is, the same sequence will form the same structure. The function of the protein is chiefly determined by its structure, because proteins whose shapes are complementary can interact with each other.

Every protein is made up of twenty kinds of amino acids which are distinguished by twenty different code letters. A protein has about two hundred amino acids on average and is represented by a linear sequence of code letters. Because every amino acid has its own properties of volume, hydrophobicity, polarity and so on, the order of the amino acids in the protein sequence gives structure and function of the protein.

The protein sequence determination technique has been so established that more than twenty thousand sequences have been specified by the letters; this number is growing day by day. The structures of proteins are also being solved. Methods such as X-ray crystallography reveal how the linear chain of amino acids fold together. But this takes so many months to solve that only three hundred protein structures have been determined so far.

An important way of discovering new genetic information is inferring the unknown structure of a protein from its sequence. We do this by analyzing the sequence of amino acids, because, fortunately, proteins that have similar sequences have similar structures. Multiple sequence alignment is one of the most typical methods of sequence similarity analysis. The alignment of several protein sequences can provide valuable information for researching the function or structure of proteins, especially if one of the aligned proteins has been well characterized.

Let us show an example of multiple sequence alignment. The next set of sequences represents four parts of different protein sequences. Each letter in the sequences means an amino acid. For instance, GDVEK stands for a row of Glycine, Aspartic acid, Valine, Glutamic acid and Lysine.

```
GDVEKGKIFIMKCSQCHTVEKGGKHKTGPNLHGLFG
ASFAEAPAGTTGAKIFKTKCAQCHTVKGHKQGNGLFG
PYAPGDEKKGASLFKTAQCHTVEKGGANKVGPNLHGVFG
PPKARAPLPPGDAARGEKLRAAQCHTANQGGANGVGYGLVG
```

A good multiple sequence alignment for the given sequences is as follows:

```
----------GDVEKG-KIFIMKCSQCHTVEKGGKHKTGPNLHGLFG
--ASFAEAPAG--TTGAKIFKTKCAQCHTV-KG--HKQG---NGLFG
------PYAPGDEKKGASLFKT--AQCHTVEKGGANKVGPNLHGVFG
PPKARAPLPPGDAARGEKL---RAAQCHTANQGGANGVG---YGLVG
          *    *         ****   *     *    *  *
```

Each sequence is shifted by gap insertion—dash characters. Each column of the resultant alignment has the same or similar amino acids. An identical pattern such as QCHT is considered to be an important site called a *sequence motif,* or simply a *motif,* because an important protein sequence site has been conservative along with evolutional cycles between mutation and natural selection. Multiple sequence alignment is useful not only for inferring the structure and function of proteins but also for drawing a phylogenetic tree along the evolutional histories of the creatures.

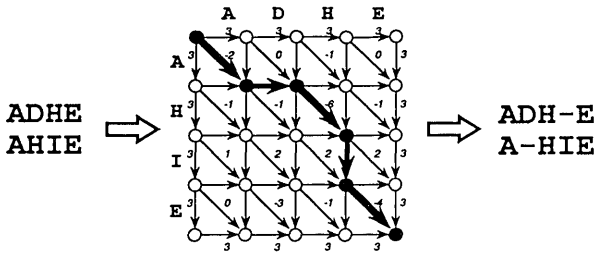


Figure 1: Pairwise dynamic programming

Computers partly solve the problem of multiple sequence alignment automatically, instead of relying on the hands and eyes of experts. The results obtained by computers, however, have not been as satisfactory as those by human experts. That is because multiple sequence alignment is one of the most time and space consuming problems. The dynamic programming algorithm [Needleman and Wunsch 1970, Smith and Waterman 1981,
Goad and Kanehisa 1982], theoretically, provides an optimal solution according to a given evaluation score. This, however, requires memory space for an N-dimensional array (where N is the number of sequences) and calculation time for the N-th power of the sequence length. Though a method was proposed to cut unnecessary computation in the dynamic programming algorithm [Carrillo and Lipman 1988], it still needs too much computation to solve any practical alignment problem. A number of heuristic algorithms for multiple alignment problems have been devised [Barton 1990, Johnson and Doolittle 1986] in order to obtain approximate solutions within a practical time. Most of these algorithms are based on pairwise dynamic programming.

Figure 1 shows the algorithm of dynamic programming applied to a tiny pairwise alignment. The algorithm searches the best path in the figurative network from the top left node to the bottom right node minimizing the total cost of arrows. Each cost indicated on an arrow reflects the similarities between the characters being compared. The best path corresponds to the optimal alignment; each arrow in the path corresponds to each column in the alignment. Vertical and

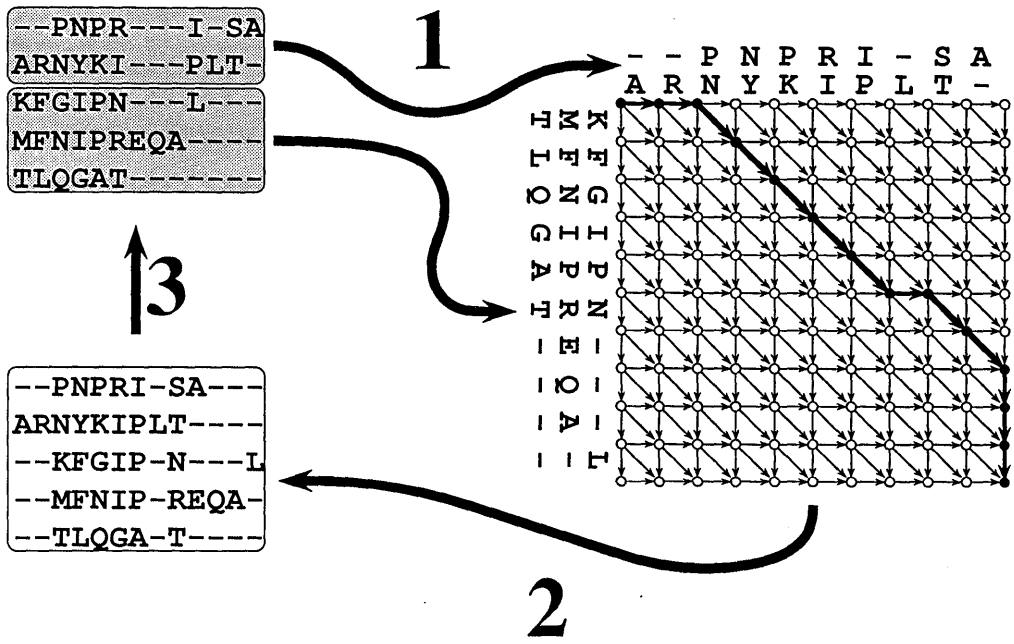


Figure 2: Iterative strategy of Berger-Munson algorithm

horizontal arrows indicate the insertion of gaps.

3 Multiple alignment system

We have developed a multiple alignment system for protein sequence analysis on PIM. The system consists of two components: a parallel iterative aligner and an intelligent refiner. The aligner uses a parallel iterative search for aligning protein sequences. The refiner uses condition-action rules for refining multiple sequence alignments given by the aligner.

3.1 Parallel iterative aligner

The search algorithm in the iterative aligner is the Berger-Munson algorithm extended in parallel. The B-M algorithm [Berger and Munson 1991] is based on the same pairwise dynamic programming method as conventional heuristic algorithms for multiple sequence alignment. The algorithm, however, features a novel randomized iterative strategy so as to generate a high-score multiple alignment.

Figure 2 illustrates the iterative strategy, whose procedure is as follows: the initially aligned sequences are randomly divided into two groups (step 1). By fixing the alignment of sequence members within each group we can optimize the alignment between the groups, using the pairwise dynamic programming method (step 2). The resultant alignment, in turn, is the starting point for the next alignment of a different pair of groups (step 3). Each iteration that improves the alignment between two sequence groups will also improve the global alignment.

Though the B-M algorithm often results in a much better multiple alignment than those obtained by conventional algorithms, its randomized iteration needs more than a few hours to solve multiple alignment of a practical scale. When a parallel machine is available, the iterative strategy extended in a parallel way is fairly helpful for reducing execution time. The B-M algorithm extended in parallel is as follows: all $2^{n-1}-1$ possible partitions of n aligned sequences are respectively evaluated by the pairwise dynamic programming method. In each iteration, the evaluation is executed in parallel and the alignment which has the best score is selected as the starting point for the next iteration.

3.2 Intelligent refiner

Aligning multiple protein sequences requires biological know-how, since the alignment score is not sufficient to evaluate them. The intelligent refiner holds dozens of condition-action rules that reflect the biological know-how for refinement. Part of the biological know-how has been obtained by interviewing human

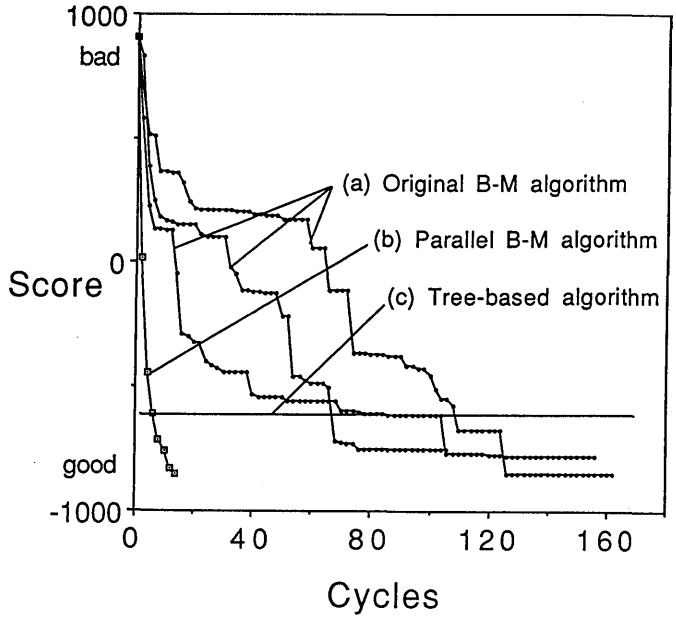


Figure 3: Comparing alignment score histories

experts. Another part of it corresponds to the information contained in a motif database *PROSITE*.

Let us explain an example of the condition-action rule, which features a well-known motif pattern called *Zinc Finger*. Zinc Finger is characterized by two separated Cs, Cysteines, and two separated Hs, Histidines. The condition part of the rule checks whether an alignment has the half-aligned motif pattern of Zinc Finger or not, and if it finds the *weak motif* pattern, it tries, in its action part, to enhance the weak pattern to make it strong (see Figure 4). Every condition-action rule is represented with a parallel logic programming language KL1.

4 Experimental results

Our multiple sequence alignment system works on PIM/m, a MIMD-type parallel machine equipped with up to 256 processing elements (PEs). We have investigated the performance of our system by testing the two components separately.

4.1 Parallel iterative aligner

The B-M algorithm enables us to gradually improve global multiple alignment. Improvement is evaluated by the alignment score. We have defined the alignment score as follows. The alignment score is a total summation of the similarity scores of every pair of aligned sequences, each of which is derived by summing up the similarity values of every character pair in the column. Each similarity value is given by the *odds matrix*. A *gap penalty* corresponding to each row of gaps in the two sequences is added to the similarity score.

We use *PAM*250 [Dayhoff *et al.* 1978] as the odds matrix, each value of which is a logarithm of the mutation probability of a character pair; zero is the neutral value. We have reversed the sign of each value of the matrix to assimilate the habit of optimization problems. So the most similar character pair, `W` *vs.* `W`, gives the lowest value, -17, and the least similar pair, `W` *vs.* `C`, gives the highest value, 8.

The gap penalty imposed on a row of k gaps is a linear relation: $a + bk$ where a and b are parameters. We set $a = 4$ and $b = 1$ as default values. The linear relation is feasible and popular for alignment done by the dynamic programming algorithm [Gotoh 1982]. Character pairs *gap vs. gap* and *outside gap vs. any character* are ignored; they are assigned the neutral value zero.

We have implemented three algorithms for comparison analysis: the original B-M algorithm, the B-M algorithm extended in parallel and the tree-based algorithm. The tree-based algorithm [Barton 1990] is one of the most typical and conventional methods for multiple sequence alignment. Figure 3 compares the histories of the alignment scores obtained by the algorithms.

```
(1)Before:
  ------------ILD---FHE-KLLHPGIQKT---TKLF--GET---YYFPNSQLLIQNIINECSICNLAKTEHRNTDM--P-TKTT
  ------------LLD---F-----LHQLTHLSFSKMKALLERSHSPYYMLNRDRTL-KNITETCKAC--AQVNASKSAVKQG-TR--
  LTDALLIT---PVLQ---LSP-AELHSFTHCG---QTAL--TLQ----GATTTEA--SNILRSCHAC---RGGNPQHQMPRGHI---
  ------VADSQATFQAYPLREAKDLHTALHIG---PRAL--SKA---CNISMQQA--REVVQTCPHC------NSAPALEAG-VN--
  ------------ISD--PIHEATQAHTLHHLN---AHTL--RLL---YKITREQA--RDIVKACKQC---VVATPVPHL--G-VN--
  ------------ILT--ALESAQESHALHHQN---AAAL--RFQ---FHITREQA--REIVKLCPNC---PDWGSAPQL--G-VN--
  (score = -781)           *   ^       ^                    ^   *  *             ^

(2)After:
  ------------ILD---F------HEKLLHPGIQKTTKLF-GET---YYFPNSQLLIQNIINECSICNLAKTEHRNTDM--P-TKTT
  ------------LLD---F-----LHQ-LTHLSFSKMKALLERSHSPYYMLNRDRTL-KNITETCKAC--AQVNASKSAVKQG-TR--
  LTDALLIT---PVLQ---LSP-AELHS-FTHCG---QTAL--TLQ----GATTTEA--SNILRSCHAC---RGGNPQHQMPRGHI---
  ------VADSQATFQAYPLREAKDLHT-ALHIG---PRAL--SKA---CNISMQQA--REVVQTCPHC------NSAPALEAG-VN--
  ------------ISD--PIHEATQAHT-LHHLN---AHTL--RLL---YKITREQA--RDIVKACKQC---VVATPVPHL--G-VN--
  ------------ILT--ALESAQESHA-LHHQN---AAAL--RFQ---FHITREQA--REIVKLCPNC---PDWGSAPQL--G-VN--
  (score = -762)           *    *        *                    ^   *  *
```

Figure 4: Application of intelligent refiner

Every algorithm solves the same small alignment problem which consists of seven sequences with eighty code letters each. The initial state of the alignment problem has no gaps inside the sequences.

(a) **Original B-M algorithm:** The randomized iterative strategy executed by a single PE is applied to the alignment problem. Each iteration cycle takes twenty-eight seconds on average.. We set thirty-two as the *convergence condition*; execution stops, if no variation of alignment score is found during thirty-two iteration cycles. Three runs with distinct sequences of random numbers give converged alignment scores: -752, -779 and -851.

(b) **Parallel B-M algorithm:** The best-choice iterative strategy executed by sixty-three PEs is applied to the alignment problem. In each iteration, sixty-three possible partitions of aligned sequences are distributed to the PEs so that they can be evaluated at the same time. Each iteration cycle takes thirty seconds on average. The execution stops if no variation of alignment score is found. The final alignment, which is obtained at the fourteenth cycle with score -851, is the same alignment as one of the three obtained in (a).

(c) **Tree-based algorithm:** The tree-based algorithm is a conventional method to rapidly produce a practical multiple alignment. The algorithm aligns sequences one after another by pairwise dynamic programming. The order in which sequences are aligned depends on the tree-like representation that was previously determined by analyzing the distance of similarity of every pair in the sequences. Our implementation of the algorithm solves the problem in eighty seconds. The alignment score of the solution, -617, is indicated by a horizontal line.

We made the following observations from these results.

1. The parallel B-M algorithm (b) solves alignment problems about ten times faster than the original B-M algorithm (a).
2. The original B-M algorithm (a) gives different alignments depending on the sequence of random numbers, whereas the parallel B-M algorithm (b) gives a constant alignment that often has a better score than obtained by (a).
3. (a) and (b) show that either of the B-M algorithms gives a much better alignment than the conventional tree-based algorithm (c).

Thus, the parallel B-M algorithm can constantly generate high-score alignments in a small number of cycles. And PIM can execute the algorithm in a practical time.

4.2 Intelligent refiner

The refiner holds dozens of condition-action rules and checks a given alignment with the condition parts in parallel. If some condition parts match the alignment, the action parts paired with the condition parts are executed so as to produce candidates for a refined alignment. After evaluation of the candidates, some of them are displayed as refined alignments. Let us show an example of the refinement.

Figure 4 (1) shows an alignment which contains a weak Zinc-Finger motif pattern. Cs are aligned completely in two columns, but Hs are not aligned completely in two columns; Q exists among identical Hs in

a column. (* indicates a completely aligned column and ^ indicates an almost completely aligned column.) Application of the intelligent refiner to the alignment produces Figure 4 (2).

The condition-action rule described in Section 3 has worked on the refinement process. The Zinc-Finger motif pattern is brought into full relief in the refined alignment. Although it has a score that is slightly worse than the previous alignment, it is a valuable alignment from a biological point of view.

Thus, the intelligent refiner helps to extract motifs from ambiguous alignment patterns and to produce biologically valuable alignments. Constructing the intelligent refiner on PIM is a profitable way, since KL1, a logic programming language on PIM, is suitable for representing such biological know-how.

5 Conclusions

We have developed a multiple sequence alignment system on PIM. The parallel iterative aligner of this system with the extended Berger-Munson algorithm can constantly generate better alignments than conventional methods in a practical time. The intelligent refiner of this system uses condition-action rules for refining alignments given by the aligner. The rules reflecting biological know-how help us to extract motif patterns from ambiguous alignment patterns. These results show that PIM is fairly available in the field of genetic information processing.

The extended algorithm searches all $2^{n-1}-1$ possibilities in parallel and selects the best one. There is a problem because the number of possibilities increases exponentially as the number of sequences grows. Some practical alignment problems with more than twenty sequences have about a million possibilities. In those cases, preprocessing with *cluster analysis* is useful for reducing the possibilities without reducing the quality of the resultant alignment. The cluster analysis divides given sequences into a few groups based on similarities between sequences; similar sequences gather in the same groups.

One of our future works is to represent complex biological know-how as a combination of simple condition-action rules.

Acknowledgments

We gratefully acknowledge Osamu Gotoh for calling our attention to the B-M algorithm. We would also like to thank Naoyuki Iwabe and Kei-ichi Kuma for providing us with the biological know-how to refine alignments.

References

[Barton 1990] J. G. Barton. Protein multiple sequence alignment and flexible pattern matching. In R. F. Doolittle (ed), *Methods in Enzymology Vol.183,* Academic Press, 1990. pp.403–428.

[Berger and Munson 1991] M. P. Berger and P. J. Munson. A novel randomized iterative strategy for aligning multiple protein sequences. *Computer Applications in the Biosciences,* **7,** 1991. pp.479–484.

[Butler *et al.* 1990] Butler, Foster, Karonis, Olson, Overbeek, Pflunger, Price and Tuecke. Aligning Genetic Sequences. *Strand: New Concepts in Parallel Programming,* Prentice-Hall, 1990, pp.253–271.

[Carrillo and Lipman 1988] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.,* **48,** 1988, pp.1073–1082.

[Dayhoff *et al.* 1978] M. O. Dayhoff, R. M. Schwartz and B. C. Orcutt. A model of evolutionary change in proteins. In M. O. Dayhoff (ed), *Atlas of Protein Sequence and Structure Vol.5, Suppl.3,* Nat. Biomed. Res. Found., Washington, D. C., 1978, pp.345–352.

[Goad and Kanehisa 1982] W. B. Goad and M. I. Kanehisa. Pattern recognition in nucleic acid sequences. I. A general method for finding local homologies and symmetries. *Nucleic Acids Res.,* **10,** 1982, pp.247–263.

[Gotoh 1982] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.,* **162,** 1982, pp.705–708.

[Iyengar 1988] A. K. Iyengar. Parallel DNA Sequence Analysis. *MIT/LCS/TR-428,* 1988.

[Johnson and Doolittle 1986] M. S. Johnson and R. F. Doolittle. A method for the simultaneous alignment of three or more amino acids sequences. *J. of Mol. Evol.,* **23,** 1986, pp.267–278.

[Needleman and Wunsch 1970] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Mol. Biol.,* **48,** 1970, pp.443–453.

[Smith and Waterman 1981] T. F. Smith and M. F. Waterman. Identification of common molecular subsequences. *J. of Mol. Biol.,* **147,** 1981, pp.195–197.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Folding Simulation using Temperature Parallel Simulated Annealing

Makoto Hirosawa,* Richard.J.Feldmann,† David Rawn,‡
Masato Ishikawa,* Masaki Hoshida,*
George Michealst

hirosawa@icot.or.jp

Abstract

We applied *temperature parallel* simulated annealing to the biological problem of folding simulation. *Water-counting* is introduced to formulate folding simulation as an optimization problem. Nobody has ever solved the folding simulation problem. We cannot obtain biologically significant consequences either. However, from the viewpoint of the evaluation value of the folding simulation, we observed the effectiveness of parallel computing.

1 Introduction

Folding simulation uses a computer to simulate the process of protein formation from its stretched state to its native folded state. This research topic has held the interest of biologists for a quarter of a century and has never been solved. No researche has been able to reach the native folded state by folding simulation. Three of the authors(Feldmann, Rawn and Micheals) have been interested in formulation for protein folding. They introduced *the water-counting model*, which requires solution by computer.

Meanwhile, the other three the authors(Hirosawa, Ishikawa and Hoshida) have studied the application of Multi-PSI [Nakajima *et al.* 1989] parallel inference machine to biological problems. A first attempt was made to the problem of multiple alignment [Ishikawa *et al.* 1991] using *temperature parallel* simulated annealing [Kumura and Taki 1990]. It was so successful that other biological applications were sought.

As the requirements of both partners matched, we combined efforts to conduct collaborative research. The purpose of this research was to investigate the applicability of the optimization algorithm, temperature parallel simulated annealing, to folding simulation and to evaluate the effectiveness of the water-counting model.

The concept of folding simulation is explained in the second section and the water-counting model and its computational formulation are introduced in the third section. Then, temperature parallel simulated annealing is explained in the fourth section. Finally, the simulation results are shown in the fifth section.

2 What is folding simulation?

2.1 Biological background of folding simulation

Proteins are biological substances and they are essential to the existence of all creatures, from humans to the AIDS virus. A protein is a linear chai of amino acids. It consists of 20 kinds of amino acids. The structure of protein is determined by the order of the amino acids in the sequence. The structure of protein is closely related to its function. Therefore, it is very important to know the structure of the protein.

Even now, it is very difficult to determine the structure of a protein. X-ray crystallography and NMR(Nuclear Magnetic Resonance) can be used to determine structure. But the former method can only be

*Institute for New Generation Computer Technology(ICOT)
†National Institutes of Health
‡Towson University

utilized when crystalization of protein is succeessful, and this crystalization is very difficult to do. The latter method can be adopted when the size of the protein is small. Both require plenty of time from months to a year.

On the other hand, we can determine the order of amino acids in the sequence of protein extremely easier than we can determine a structure of a protein. A technique for determining the sequence of a protein has been established. That is why folding simulation is important and necessary.

Folding simulation simulates, by computer, the process of protein formation from its stretched state to its native folded state. Before simulation starts, information on the order of the amino acids is provided.

2.2 Folding simulation as an optimization problem

Folding simulation is a research topic that has fascinated hundreds of theoretical bio-chemists for a quarter of a century. The molecular dynamic method is, theoretically, able to solve folding simulation problem. The method precisely simulates the movement of each atom driven by kinetic forces. However, it requires such huge amounts of computational time that actual folding simulation problems cannot be solved (it can simulate pico-second movements of a protein whereas the whole folding process takes a few seconds or more). To make the computational time tractable, we have to seek effective approximation methods.

In each approximation method, abstract representation (*e.g. the amino-acid ball* which represents all atoms in an amino acid as a single ball) and the limited structure state (*e.g.* limited location or angle) are often introduced. We can regard such an approximation method as combinatorial optimization, because each discrete state is evaluated by a properly-defined potential energy to be minimized and effective transition between states is devised.

One of the most frequently employed approximation methods is lattice representation [Ueda *et al.* 1978] [Skolnick and Kolinsky 1991], which restricts the position of amino acids in 3-dimensional lattice cells. Although the lattice representation can remarkably reduce computational time for folding simulation, no significant result had been produced until recently. That is partly because the lattice formulation might not be good enough to simulate the folding process, and partly because the computational power required might have still been too big.

The work by Skolnick and his co-researchers is the first research that did not poorly reproduce the native structure of protein by the folding simulation. However, the parameterization he employed to reproduce the native structure has drawn criticism.

3 Computational Formulation of Folding Simulation

We introduced a water-counting model to approximate folding simulation concisely and to formulate folding simulation as an optimization problem based on the model. The water-counting model employs a lattice representation for protein and water.

3.1 Concept of water-counting model

Coming back to basic biological knowledge, we have sought a simulation method that requires the most minimal parameterization possible. Then, we found the water-counting model as a biological model.

In 1958, I.M. Klotz recognized that the folded structure of a protein depends upon its interaction with water [Koltz 1958]. At about the same time, W.Kauzmann showed that the *hydrophobic effect* provides the principle driving force for protein folding [Kaumann 1959].

Hydrophobicity is a measure that represents the degree to which amino acids don't favor water. Amino acids that favor water are called hydrophilic amino acids, and those that don't are called hydrophobic amino acids. Hydrophobic force is caused by the above tendency of amino acids. Since many biologists today, if not all, still recognize hydrophobic force as a primary force, we simplified folding simulation by employing hydrophobic force without using any other kind of force.

Next, we investigated the origin of hydrophobic force. We concluded that the binding and detaching of water to and from amino acids produces hydrophobic force. We can interpret the global minimum energy of protein in terms of the number of water molecules bound to proteins. Because the energy is calculated by the number of water molecules around amino acid, we

coined it the water-counting model.

3.2 Representation of Protein

We will describe a way to represent protein on a 3-dimensional lattice. In lattice cells, any place protein is not present, water will fill.

As described earlier, proteins are linear chains of amino acids. Each amino acids is composed of two parts, namely, a main chain and a side chain. Main chains form the backbone of protein. Side chains of amino acids determine the properties of the amino acid.

The main chain of an amino acid serves to connect adjacent amino acid. The relative location between two adjacent amino acids is like the move that a knight in chess makes, but on a 3-dimensional lattice (Figure 1), $(\pm 3, \pm 1, \pm 1)$. Every main chain of amino acid occupies 27 $(= 3^3)$ lattice cells.

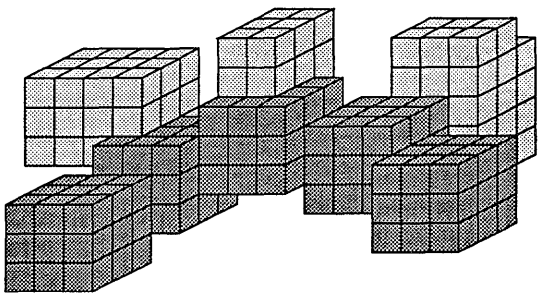


Figure 1: Representation of a part of protein: main chains(shaded) and side chains(unshaded)

Each of the twenty kinds of amino acids has different side chains (Table 1). For example, their volume (the number of lattice cells occupied) and hydrophobicity [Janin 1979] differ.

3.3 Evaluation of State

The energy of states are evaluated in the following formula.

$$E(Energy) = \sum_m^{sidechains} (Water\ Count_m - 1) \times Hydrophobicity_m$$

$$Water\ Count_m = \frac{Number\ of\ adjacent\ cells\ (of\ side\ chain)\ occupied\ by\ other\ amino\ acids}{The\ number\ of\ adjacent\ cells\ of\ the\ side\ chain}$$

In the first formulas, the terms from hydrophobic amino acids are negative and those from hydrophilic amino acid are positive. The more the absolute value of the hydrophobicity of an amino acid is, the greater its contribution to the energy is.

Amino acid	A	C	D	E	F	G	H
Hydrophobicity	0.3	0.9	−0.6	−0.7	0.5	0.3	−0.1
Volume	12	16	20	32	52	0	40

Amino acid	I	K	L	M	N	P	Q
Hydrophobicity	0.7	−1.8	0.5	0.4	−0.5	−0.3	−0.7
Volume	48	60	48	40	28	28	40

Amino acid	R	S	T	V	W	Y
Hydrophobicity	−1.4	−0.1	−0.2	0.6	0.3	−0.4
Volume	68	16	28	36	68	56

Table 1: Characteristics of amino acid side chains: each letter signifies one of 20 amino acids, for example, E signifies glutamic acid.

The energy can be reduced both by increasing the amount of water around the hydrophilic amino acid and by reducing the amount of water around the hydrophobic amino acid. The minimization of energy has the effect of inviting hydrophobic amino acids toward the center of the protein where there is less water and to oust hydrophilic amino acids to the surface of the protein where water is abundant.

3.4 Transition between States

As a transition from one state to another, we introduce two classes of transition. One is rotational transition, and the other is translational transition (Figure 2).

Rotational transition is the move that proteins probably take in actual folding processes. We first focus on one amino acids and select which side of the protein to rotate (in the figure, the right side is selected). Then, by regarding a connection line between the focused amino acid and its adjacent amino acid (in the figure, the adjacent amino is on the left) as an fixed axis, the selected side of the protein is rotated.

Translational transition is the moving of proteins that is done for computational convenience. As with rotational transition, one amino acid is focused on and the side to move is selected. Then, the adjacent amino acid of the selected protein is moved and other amino acids on the selected side are moved translationally. (the direction to translate is specified by the move of the adjacent amino acid).

After a new state is created by the transition se-

lected, a collision check is executed. If, in the next possible state, there is no multiply occupancy of any lattice cell by different parts of the protein, this state is acceptable. Otherwise, the state is discarded and new transitions are tested until some that is accepted is found.

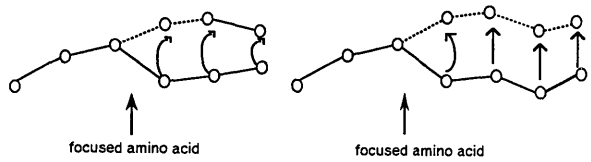


Figure 2: Rotational transition (left) and Translational transition (right)

4 Temperature Parallel Simulated Annealing

In the proceeding section, we formulated folding simulation as the problem to search for the minimum energy in a solution space. We employed temperature parallel simulated annealing as an algorithm to find a global optimal solution. Temperature parallel simulated annealing is an algorithm that can circumvent a scheduling problem of simulated annealing (SA), by introducing the concept of parallelism in temperature.

In this section, SA is explained firstly, then temperature parallel SA is introduced.

4.1 SA

SA is a stochastic algorithm used to solve complex combinatorial optimization problems [Kirkpatrick 1983]. It searches for a global optimal solution in a solution space without being captured in local optima.

SA simulates the annealing process of physical systems using a parameter, *temperature*, and an evaluation value, *energy*. At high temperatures, the search point in the solution space jumps out of local energy minimum. At low temperatures, the point falls to the nearest local energy minimum.

An outline of the SA algorithm is as follows. Given an arbitrary initial solution x_0, the algorithm generates a sequence of solutions $\{x_n\}_{n=0,1,2,\ldots}$ iteratively, finally outputting x_n for a large enough value of n. In each iteration, the current solution x_n is randomly modified to get a candidate x'_n for the next solution, and the variation of the energy $\Delta E = E(x'_n) - E(x_n)$ is calculated to evaluate the candidate. When $\Delta E \leq 0$, the modification is good enough to accept the candidate: $x_{n+1} = x'_n$. When $\Delta E > 0$, the candidate is accepted with probability $p = \exp(-\Delta E/T_n)$, but rejected otherwise: $x_{n+1} = x_n$, where $\{T_n\}_{n=0,1,\ldots}$ is a *cooling schedule* (a sequence of temperatures decreasing with n).

Because solution x_n is distributed according to the Boltzmann distribution at temperature T, the distribution converges to the lowest energy state (optimal solution) as the temperature decreases to zero (Figure 3). Thus, one might expect SA to be capable of providing the optimal solution, in principle. It is well-known that the cooling schedule has great influence on SA performance. This is where the *cooling schedule problem* arises.

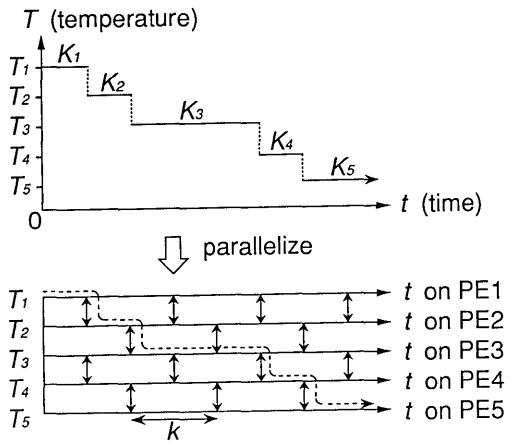


Figure 3: Ordinary SA and temperature parallel SA

4.2 Temperature Parallel SA

The basic idea behind the algorithm is to use parallelism in temperature [Kumura and Taki 1990], to perform annealing processes concurrently at various temperatures. The algorithm automatically constructs an appropriate cooling schedule from a given set of temperatures (Figure 3). Hence, it partly solves the cooling schedule problem.

The outline of the algorithm is as follows. Each processor maintains one solution and performs the annealing process concurrently at a *constant* temperature that differs between processors. After every k annealing steps, each pair of processors with adjacent temperatures performs a *probabilistic exchange of solutions*. Let $p(T,E,T',E')$ denote the probability of the exchange between two solutions: one with energy E at temperature T and the other with energy E' at temperature T'. This is defined as follows:

$$p(T,E,T',E') = \begin{cases} 1 & \text{if } \Delta T \cdot \Delta E < 0 \\ \exp(-\frac{\Delta T \cdot \Delta E}{TT'}) & \text{otherwise} \end{cases}$$

where $\Delta T = T - T', \quad \Delta E = E - E'.$

The probability has been defined such that solutions with lower energy tend to be at lower temperatures. Hence, the solution at the lowest temperature is expected to be the best solution so far. The cooling schedule is invisibly embedded in the parallel execution.

The temperature parallel algorithm has advantages other than the dispensability of the cooling schedule. We can stop the execution at any time and examine whether a satisfactory solution has already been obtained.

The algorithm of temperature parallel SA is implemented as a tool kit. When we want to solve some problem using temperature parallel SA, if we use the tool kit, all we have to do is to write a program that just corresponds to the problem.

5 Experiment and Discussion

We selected flavodoxin, whose structure is known, as the protein to simulated. This protein is of a medium size and has 138 amino acids. We ran the folding simulation program using temperature parallel SA on Multi-PSI using 20 processors over 10 days. This corresponds to 30,000 cycles. We also ran the folding simulation program using *simple parallel* SA in 30,000 cycles, also with 20 processors.

The simple parallel SA is a naive combination of sequential SAs: every available processor has one solution and anneals it sequentially using a *distinct* sequence of random numbers. All resultant solutions are compared with each other and the best one, the one with the minimum energy value, is selected as a solution for the algorithm .

5.1 Experimental result

The minimum energy versus the cycles of simulation of those two algorithms is plotted in Fig.4. In the figure, the result using *sequential* SA, ordinary SA, is also plotted. Its energy is the average of energy obtained by sequential SAs in the simple parallel SA.

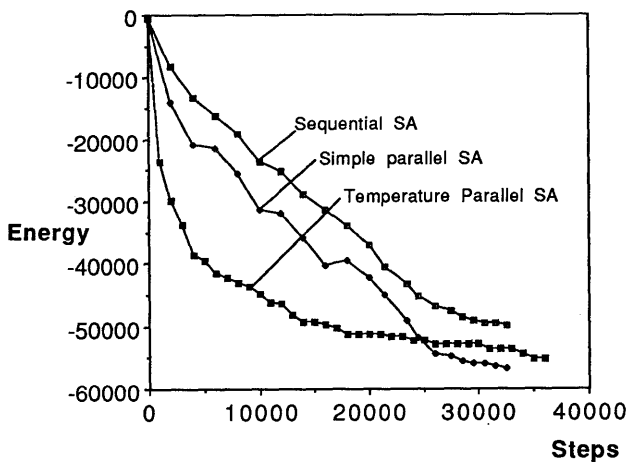


Figure 4: Energy history of folding simulation

One of the structure of flavodoxin produced by the program is shown in Figure 5. Unfortunately, its structure is not similar to the structure of real flavodoxin. However, a favorable tendency ,where hydrophobic amino acids are inside the structure while hydrophilic amino acids are outside the structure, was observed.

5.2 Discussion

The effectiveness of the water-counting model will be evaluated first, then the effectiveness of the temperature parallel SA as an optimization method for practical problems will be evaluated.

The structure of the flavodoxin produced was not similar to its real structure. However, this doesn't necessarily indicate a defect in the water-counting model. We, instead, think that the result is due to insufficiency of transitions we introduced.

The rough structure of protein, especially that of small protein, can be reproduced by global transition

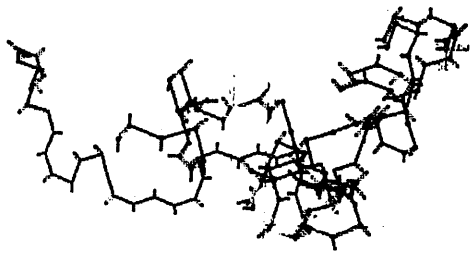


Figure 5: Result structure of folding simulation (flavodoxin)

that is like rotational transition and translational transition. There is little collision among amino acids in the path from the stretched state to the roughly formed structure. However, a fine protein structure is rarely reproduced by global transitions alone due to the collisions.

We think that the local transition modes that can avoid collision should be incorporated to reproduce the native structure with collision check. We are planning to introduce a local transition, kink mode [Skolnick and Kolinsky 1991]. We think that the necessary mode of transition must be incorporated before we can evaluate the effectiveness of the water-counting model.

Next, we evaluate the effectiveness of temperature parallel SA as an optimization method by using Figure 4. Readers who are familiar with SA should consult Appendix.

We made the following observations from this energy profile in consideration of the above points.

1. Two kinds of parallel SAs made better results within a fixed time than sequential SA. This is simply the effect of multiple processors.

2. Up to the middle stage of simulation, temperature parallel SA is always better than simple parallel SA. This is because temperature parallel SA can produce optimal solutions as that time.

3. Two kinds of parallel SAs have almost the same final energy value.

Figure 4 shows the tendency for energy of all methods to be minimized further after the completion of a specified cycle of simulation. Only simulation by temperature parallel SA can be resumed without rescheduling. Because two kinds of parallel SAs are almost the same, we think that temperature parallel SA is more advantageous than simple parallel SA.

Simulated annealing is most effective when states generated at higher temperatures can cover nearly all the solution space. In the case of folding simulation, this is hard to do it. We are now engaged in trying to restrict the solution space of simulated annealing by knowledge and/or heuristics to the extent that the solution space can be covered by simulated annealing.

6 Summary

We studied folding simulation as an application of parallel simulated annealing. This program was written in KL1 and was executed on the parallel inference machine Multi-PSI. As the biological model the water-counting model that uses lattice representation and only hydrophobic interaction between amino acids was selected.

The structure of flavodoxin produced by program is not appropriate from a biological point of view. This suggests that the program requires further improvements. The kink mode of transition is one candidate to incorporate.

However, the insight was gained from the point view of computer science, namely evaluation of temperature parallel simulated annealing. The result using temperature parallel SA had almost the same final energy value (which is much better than that obtained by sequential SA) as the result using simple parallel SA. In consideration of the dispensability of rescheduling when further optimization is necessary, temperature parallel SA was proved to be advantageous.

The other thing we learnt was that a module that restricts the solution space of folding simulation is required. We think knowledge engineering must be employed to do this, and also that KL1 is suitable for use.

Acknowledgment

The authors would like to especially thank Y. Totoki of IMS for his programming and experimentation with the

folding simulation. Without his endeavors which were conducted through what should have been his winter vacation, this paper couldn't have been written. We would also like to thank Kouichi Kimura, the founder of temperature parallel SA, for valuable discussions.

We also thank Dr. Uchida and Dr. Nitta for their support in this international collaboration.

References

[Kumura and Taki 1990] Kimura, K. and Taki, K. (1990) Time-homogeneous parallel annealing algorithm. *Proc. Comp. Appl. Math. (IMACS'91),* **13,** 827–828.

[Nakajima *et al.* 1989] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T. (1989) Distributed implementation of KL1 on the Multi-PSI/V2. *Proc. 6th Int. Conf. on Logic Programming.*

[Ishikawa *et al.* 1991] Ishikawa, M., Hoshida, M., Hirosawa, M., Toya, T., Onizuka, K. and Nitta, K. (1991) Protein Sequence Analysis by Parallel Inference Machine. *Information Processing Society of Japan, TR-FI-23-2,* (in Japanese).

[Gierasch and King 1990] Gierasch, L.M and King, J(ed) (1990) Protein folding. *American association for the advance of science.*

[Skolnick and Kolinsky 1991] Skolnick, J. and Kolinski, A. (1991) Dynamic Monte Carlo Simulation of a New Lattice Model of Globular Protein Folding, Structure and Dynamics. *Journal of Moleculer Biology Vol. 221 no.2,* 499-531.

[Ueda *et al.* 1978] Ueda, Y., Taketomi, H. and Go, N. (1978). Studies on protein folding, unfolding and fluctuations by computer simulation. A three dimensional lattice model of lysozyme. *Bilpolymers Vol.17* 1531-1548.

[Koltz 1958] Koltz,I.M. (1958) *Science Vol.128,* 825-

[Kaumann 1959] Kauzmann (1959) *Advances in Protein Chemistry, Vol.14, no.1.*

[Janin 1979] Janin, J. (1979) Surface and side volumes in globular proteins. *Nature(London) Vol.277,* 491-492.

[Kirkpatrick 1983] Kirkpatrick, S., Gelatt, C.D. and Vecci, M.P. (1983) Optimization by simulated annealing. *Science, vol.220, no.4598.*

Appendix

Readers should pay attention to the following possibility when they discuss the result of Figure 4.

1. The two energy histories obtained by the two parallel SA algorithms might include the influence of statistical fluctuation, because each parallel algorithm was experienced only once. The sequential algorithm, however, was done twenty times and each point in the history represents the average energy value.

2. All SA procedures may be quick quenched instead of annealed, because the number of steps at each temperature, 1500, would be relatively small against the size of the solution space. If so, temperature parallel SA is disadvantageous for obtaining good energy in a short time, because not all processors in temperature parallel SA will necessarily do quick quenching; some processors may often do real annealing.

3. All SA proceduresy may not reach any minima in the solution space, because every decline in energy history is not sufficiently saturated.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Toward a Human Genome Encyclopedia

Kaoru Yoshida [1,1], Cassandra Smith [2],
Toni Kazic [3], George Michaels [4], Ron Taylor [4],
David Zawada [5], Ray Hagstrom [6], Ross Overbeek [6]

[1] Division of Cell and Molecular Biology, Lawrence Berkeley Laboratory, Berkeley, CA 94720, U.S.A.
[2] Department of Molecular and Cell Biology, University of California at Berkeley, Berkeley, CA 94720, U.S.A.
[3] Department of Genetics, Washington University School of Medicine, St. Louis, MO 63110, U.S.A.
[4]Division of Computer Research and Technolgy, National Institutes of Health, Bethesda, MD 20894, U.S.A.
[5]Advanced Computer Applications Center, Argonne National Laboratory, Argonne, IL 60439-4832, U.S.A.
[6]Division of Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439-4832, U.S.A.

Abstract

Aiming at building a human genome encyclopedia, a human genome mapping database system, *Lucy*, is being developed. Taking chromosome 21 as the first testbed, more than forty maps of different kinds have been extracted from publications, and several public and local genome databases have been integrated into the system. To our knowledge, Lucy is one of the first systems that have ever succeeded in genome database integration. The success owes to the following key design strategies: (1) A sequential logic programming language, Prolog, has been used so that the database construction and query management could rely on the internal database facility of Prolog. (2) An object-oriented data representation has been employed, so that any kind of data could be manipulated in the same manner. (3) A mini language, *map expression*, has been designed, which enables map representation in a relative-addressing manner and also linkage of one map to another. These strategies are applicable for building a genome mapping database not only on human chromosome 21 but also beyond chromosomes and beyond species.

1 Introduction

1.1 Why Biological Applications?

The fact that only four DNA bases (adenine, thymidine, guanine, and cytosine – symbolically represented as A, T, C and G respectively) encode most of the information on current life and its history is fascinating from the viewpoint of computer science. More interesting is that many biological reactions are due to the property that A and T make a complementary pair as well as G and C do. Genome analysis is potentially a large application area for symbolic computation. As biological experimental methodology develops, more gene information is accumulated and analysed. This holds especially true for such large scale models as the human genome whose total genome size reaches a few billion of bases. Since NIH (National Institute of Health) and DOE (U.S. Department of Energy) embarked a joint national research initiative [30, 31], human genome projects have been initiated in many other countries and research activities are being expanded and accelerated day by day [89, 65, 83]. To proceed efficiently in the ever accelarating climate of current biological research, strong support and feedback from computer-aided analysis is mandatory [74, 53, 39].

1.2 What is a Physical Mapping Process?

Genome mapping is similar to geographical mapping. The genome mapping is now akin to the early times of geography. First of all, it is not known yet exactly how big the genome is. Continents, countries, states, cities and streets work as *geographical markers* which give positional information, *addresses*, on the earth. As well, continental-level landmarks with location-specific information such as a single copy DNA sequence (i.e., sequences that occur only once in the genome) [70] have been discovered here and there on the genome; fragmentary maps around these landmarks are being drawn, some of which are being glued one to another. Furthermore, as there are geographical maps and time-zone maps, there are different kinds of genome maps, roughly categorized into two kinds: physical maps giving physical distances (i.e., the number of bases lying) between the markers and genetic maps giving recombination frequencies between the markers. This section introduces what is genome physical mapping. It should help in understanding the genomic data which will be involved in the genome mapping databases described in later sections. For more details, consult [90].

Chop, Identify and Assemble. Figure 1 shows how physical mapping is done. In general, a genome is too large to be directly sequenced[2] with the current sequencing technology. For example, the total size of hu-

[2] *DNA sequencing* means experimentally reading a DNA sequence consisting of A, T, C, G bases.

man chromosome 21, the shortest human chromosome, is thought to be 50 to 65 Mb (mega bases), while the maximum length of DNA read per day is about 500 bases, including reading error corrections, and the cost of sequencing is about one dollar per base [44, 9]. If the chromosome 21 were read in a serial manner, it would take 250 years. Hence, first of all, the chromosome has to be excized into pieces (called *fragments*), which are small enough for further analysis, such as a 20-30 Kb (kilo bases) to a 2-3 Mb (*step (1)*). The excision is done by a physical method (e.g. by irradiation [27]) or by a chemical method (e.g. by digestion with restriction enzymes[3]) (*step (3)*). Then, the DNA fragments are to be assembled to the original chromosome. For the assembling, there are a variety of methods, depending on (a) whether or not the DNA fragments may overlap, (b) how the overlap and adjacency of the DNA fragments are detected, and (c) whether each step of experiment is attempted against an individual fragment or to a group of fragments at a time.

A Conventional Physical Mapping Method. Figure 1 shows a conventional mapping method which deals with non-overlapping restriction fragments. This method starts with chemical digestion using a restriction enzyme. The restriction fragments are sorted in size (by the electrophoresis method (*step (4)*) and assigned to an approximate region through hybridization[4]. Each fragment is hybridized to a variety of cell lines[5]. As each cell line covers a different region, the pattern of hybridization signals against different cell lines determines which region the target fragment resides (*step (2)*). Next, hybridization is attempted against probes within that region. With a positive hybridization signal on with a probe, the fragment is determined to lie *around* the address of the probe (*step (5)*).

A clone containing a specific restriction site is called a linking clone. A linking clone is split at the restriction site and then each half is hybridized to complete digests[6]. As the two halves are known to be next to each other, complete digests fished by the halves of a linking clone are found to be adjacent. Thus, linking clones introduce the notion of adjacency that works as a strict constraint in linear-ordering restriction fragments.

As a result of hybridization against a number of probes, fragments are eventually given a linear order (*step (6)*). The process (3) thru (6) is repeated until a map with the desired precision is obtained.

A New Physical Mapping Method. A new method, called *clone contig assembly*, is shown in Figure 2. This method uses clones of overlapping fragments of almost the same size determined by the cloning vector. By determining overlapping pairs of clones, walking is attempted from one clone to another. The resulting walking path forms an island of contiguous clones, that is called a *contig*. This method has variations depending on how the overlaps are detected (e.g., whether based on the restriction digest pattern of each clone or based on hybridization signals [54]). Furthermore, the overlap detection can be attempted against a group of clones at a time, in common. The feasibility of extracting the maximum amount of information in every step of biological experiment and the potential for automation are attracting much attention to these contig assembly methods [29]. In addition, given a set of overlapping clones, the variation of length and overlaps of clones gives a statistical limit on the number of independent islands which can be constructed from the clones [69, 26, 52]. It should be noted that this method can be carried out, vigorously relying on statistical and computational analysis [14, 37].

In summary, the physical mapping process consists of the three steps: (1) excising the whole DNA into pieces, (2) characterizing every piece through hybridization or digestion, and (3) assembling the pieces. While steps (1) and (2) are done through biological experiments, step (3) is a probablistic combinatorial problem. In order to solve this problem, information retrieval from a variety of genome databases is required together with powerful computational tools.

1.3 Mapping Data and Mapping Knowledge

Section 1.2 introduced the physical mapping process from the viewpoint of biological experiments. The resulting data are published in the form of inventories as shown in Tables 1 and 2.

Identification and Adjacency. Table 1 gives a relation between hybridization probes and restriction fragments obtained by digesting cell line WAV-17 with restriction enzyme *NotI* [88]. For instance, row 1 implies that clone 231C which is a representative of locus D21S3 hybridizes to a 2200Kb complete digest and to two partial digests: a 2200Kb fragment and a 2600Kb fragment.

Section 1.2 introduced linking clones with the notion of adjacency. HMG14l and HMG14s are the two halves

[3] *Restriction enzymes* recognize some specific DNA pattern of four to a dozen of bases and cut a double-stranded DNA at some specific position in the pattern.

[4] A double stranded DNA is formed if each strand contains a complementary sequence to the other. Hybridization is an attempt to make a double-stranded DNA or an RNA-DNA hybrid using this property. By labelling a probe (i.e. the counterpart) with an isotope or a dye, by means of autoradiograph or florescence one can detect if the probe has hybridized to the target or not.

[5] *Cell lines* are DNA segments which are generated by deleting a portion of chromosomes or by translocating between different chromosomes.

[6] *Complete digests* are restriction fragments obtained when the restriction enzymes react to completion, i.e., everyone of the target sites is cut. In contrast, *partial digests* are those which contain some fraction of the target sites uncut.

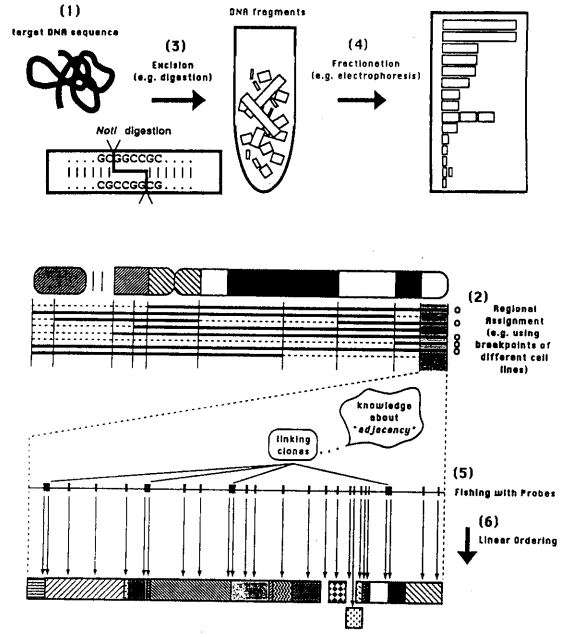


Figure 1: A process of restriction map construction

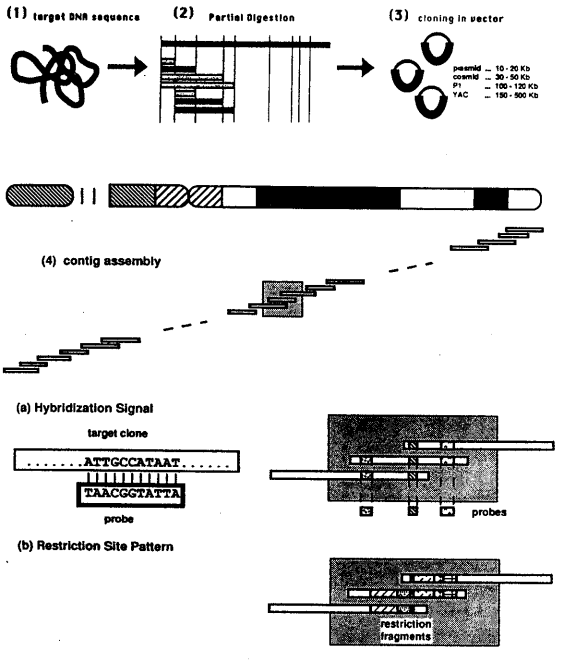


Figure 2: A process of contig assembly

Table 1: Restriction fragments and hybridization probes

	Probe		*Not*I restriction fragments
	locus/gene	clone	
1	D21S3	231C	2200,2600
2	HMG14	HMG14l	75
3	HMG14	HMG14s	300,360,560,630
4		6-40-3	300,360,530,1000
5		D13s	300,2100,2900
6		D13l	75,1800
7	*D21S101	JG373	1800,2100,2300,2600
8	*D21S15	pGSE8	2000,2400,2700
9		LA171l	1800
10		LA171s	750,2100,2350
11	*D21S51	SF93	750,1200,1800,2050,2300
12	*D21S53	512-16P	750
13	*D21S39	SF13A	750
	:	:	

of linking clone HMG14. Hence, the 75Kb fragment hybridized to HMG14l must be next to the 300Kb fragment hybridized to HMG14s. Similarly, for a pair of D13s and D13l, the 300Kb fragment and the 75Kb fragment must be adjacent; for another pair of LA171l and LA171s, the 1800Kb fragment and the 800Kb fragment must be adjacent. The 300Kb fragments in rows 3 to 5 can be interpreted to be identical, assuming that the 360Kb fragment (in rows 3 and 4) be a partial digest containing the 300Kb and the 75Kb fragments and also assuming that the 2100Kb fragment (in rows 4 and 5) be a partial digest containing the 300Kb and 1800Kb fragments.

Thus, given a relationship of restriction fragments and hybridization probes, each restriction fragment is identified using strict constraints such as linking clones and also using its neighborhood information such as a pattern of partial digests.

Confirming Information. In Table 1, the 750Kb fragments in rows 10 to 13 seem to be identical. Also, the ordering of loci D21S101 (row 7) and D21S15 (row 8) is not evident in this table, nor the ordering among loci D21S51, D21S53 and D21S39.

Table 2 shows a relationship of multiple kinds of restriction fragments (of a different cell line, CHG3) and hybridization probes [17] around the same region as Table 1. With an assumption that the *Not*I restriction sites be rather conserved in different cell lines and considering of 10-20% errors in size, the 750Kb fragment in Table 1 can be interpreted to correspond to the 700Kb fragment in rows 4 to 6 in Table 2. The identification of the 750Kb fragments in Table 1 is confirmed by the same set of *Mlu*I digests (<200Kb, 1250Kb and 1400Kb) and *Nru*I digests (600Kb and 2000Kb) found around the 700Kb fragment in Table 2. As for the ordering of loci D21S101 and D21S15, the 1600Kb *Mlu*I fragment in rows 1 and 2 connects D21S101 with D21S3 and the 1400Kb *Mlu*I

Table 2: Multiple digests

	Probe		Restriction fragments		
	locus/gene	clone	*Not*I	*Mlu*I	*Nru*I
1	D21S3	pPW231	700,1800	700,1600	600
2	D21S101	JG373	1400	1000,1600	1400,2000
3	D21S15	E8	1400	1250,1400	1400,2000
4	D21S39	SF13A	700	<200,1250,1400	600,2000
5	MxA/B		700	<200,1250,1400	600,2000
6	D21S51	SF93	700	<200,700,1400	500
:	:	:	:	:	:

fragment in rows 3 to 6 connects D21S15 with D21S39.

In general, mapping data contain a non-trivial imprecision which clouds their interpretation. Interpretation for a set of mapping data becomes less ambiguous with additional information. It is obviously efficacious to accumulate data until a convincing interpretation is acquired.

1.4 Genome Mapping Databases

Public Databases and Laboratory Notebooks. The constantly growing population of genome databases [80] contains precious few mapping databases even considering different species, such as mouse [47], *Caenorhabditis elegance* [32] and *Eschericia coli* [6]. As for human, GDB (Genome Data Base) [40] is the only public mapping database. It contains information about genes, loci (landmarks), clones, contacts and maps. As for maps, consensus maps are collected each of which contains merely the consensus order of loci, without information on physical/genetic distance between loci yet [75].

Laboratory data that are primary or secondary level of experimental data including image films will someday be available in so-called *laboratory notebook databases* which are now under development [54, 68, 4, 58]. Especially for the contig assembly mapping method for which a computer analysis environment is essential, system development efforts are intensive and have been applied for mapping chromosomes X, 16 and 19 [54, 23, 66].

What seems to be missing in genome databases is a continuous link between public databases and laboratory notebook databases. There is a strong need to compare laboratory mapping efforts against those reported in publications and public databases.

Implementations, Interfaces and Integrations. In terms of implementation strategies, most genome databases, including the above, have been implemented using relational database management systems which are based on a *normal form* (or *flat*) relational model [24]. Also these databases provide a query language (usually SQL) interface for programmers and an interactive window interface for end users, both of which rather directly reflect the underlying implementation. Programmers and users must be knowledgable about implementation issues, such as how each relational table is linked to others.

A high level interface is also required for easily sharing and exchanging data between different databases. Among leading database integration efforts, GenInfo [71, 60] is notable. Three databases: Genbank (DNA sequence database), PIR (protein sequence database) and MEDLINE (medical/biological literature database) are converted into the form of an object-oriented data representation language, ASN.1[7] [72], so that data can be easily exchanged among the databases. ASN.1 has also been applied to the construction of a metabolic compound database [49].

In summary, various kinds of information are involved in the genome mapping process. The integration of different databases is a key issue in proceeding further biological research.

1.5 Goal and Strategies

Many queries issued in the physical mapping process are imprecise, e.g., "Get all information around this locus", and "What are the consensus and differences around this locus in all collected maps?" To address these queries, all related information must first be collected from publications and various databases into a map in which all available information is woven at every location of human genome, i.e. a human genome encyclopedia. Then, using this encyclopedic map, a genomic grammar [81] will interface to the user.

The construction of a human genome database system, *Lucy*[8], has started. Taking chromosome 21 as the first testbed, more than forty maps of different types have been collected from publications, and several public and local databases have been integrated into the system. Currently, the system is ready to answer rather general queries such as shown above. To our knowledge, this is the first integrated physical mapping database that has ever been implemented.

The key design features which have enabled the prototyping of Lucy are:

- logic programming,
- object-oriented data representation and query interface,
- map representation language.

The following sections will describe each of these features in detail.

[7] Abstract Syntax Notation 1, ISO 8824.

[8] The name is derived from the nickname given to the first fossile of hominid [48]. The motto herein is *"For any question on human, ask Lucy"*.

2 Representation of Genome Information

2.1 Exploitation of Logic Programming Featuers

Lucy has been implemented in a sequencial logic programming language, Prolog, for its following features:

1. **Database Facility and Inference Mechanism:** Its internal database facility and inference mechanism enable validation of biological data and rules as knowledge immediately when they are expressed as Prolog predicates (programs). Even if they were expressed as Prolog terms (data) as second order predicates, the inference mechanism could be implemented rather easily in Prolog.

2. **Declarative Expresssion and Set Operations:** Its declarative expression and (built-in) search utilities (e.g., built-in set operations such as `setof` and `bagof`) minimize the amount of programming effort for knowledge representation and database retrieval.

3. **Recursive Queries:** Its capability of handling recursive programming and recursive data structures enables a straightfoward implementation of recursive queries that are hard to be implemented with normal form relational databases and conventional query languages such as SQL [28].

4. **Foreign Language Interface:** It is necessary to have a foreign language interface (which is provided in several Prolog implementations) to other conventional but efficient languages, such as C and Fortran, in order to import and develop the computationally intensive sequence analysis and statistical mapping tools.

5. **Portability:** Lucy should be developed as a real system to be used for biological analysis. The stability and portability of the system are the first priority.

2.2 Object-Oriented Representation and Interface

The hybridization results shown in Tables 1 and 2 in Section 1.3 could be represented as Prolog facts of a flat relational form as follows:

```
%-------------------------------------------------------------------
% table1(Locus,    Clone,    NotIdigests).
%-------------------------------------------------------------------
  table1('D21S3',  '231C',   [2200,2600]).
  table1('HMG14',  'HMG14l',  [75]).
  table1('HMG14',  'HMG14s', [300,360,560,630]).
```

```
%-------------------------------------------------------------------
% table2(Locus,     Clone,    NotIdigests, MluIdigests,  NruIdigests).
%-------------------------------------------------------------------
  table2('D21S3',   'pPW231', [700,1800],  [700,1600],  [600]).
  table2('D21S101', 'JG373,   [1400],      [1000,1600], [1400,2000]).
  table2('D21S15',  'E8',     [1400],      [1250,1400], [1400,2000]).
                :
```

Then, for every element involved in these tables, such as loci, clones and enzymes, information collected from publications and public databases was stored similarly in a flat relational form. Obviously, as the number and variety of relations increased, it will be accordingly difficult to program and maintain the database in this format, and to remember the exact form of each relation.

Another burden handling various different tables becomes obvious when encoding mapping rules. For example, the following program defines the notion of adjacency introduced with linking clones, namely that two restriction fragments are adjacent if one fragment is hybridized to one half linking clone and the other fragment to the other half linking clone and if the restriction fragments are both complete digests:

```
is_adjacent_to(FragmentA, FragmentB) :-
    is_half_linking_clone(HalfLinkingCloneP, LinkingClone),
    is_half_linking_clone(HalfLinkingCloneQ, LinkingClone),
    HalfLinkingCloneP \= HalfLinkingCloneQ,
    is_hybridized_to(HalfLinkingCloneP, FragmentA, Enzyme),
    is_hybridized_to(HalfLinkingCloneQ, FragmentB, Enzyme),
    FragmentA \= FragmentB,
    is_complete_digest(FragmentA),
    is_complete_digest(FragmentB).
```

Here troublesome is that if hybridization results were stored in various forms, predicate `is_hybridized_to/3` would have to be defined for each kind of digests in each different table, as follows:

```
is_hybridized_to(Probe, Fragment, 'NotI') :-
    table1(_, Probe, NotIFragments),
    member(Fragment, NotIFragments).

is_hybridized_to(Probe, Fragment, 'NotI') :-
    table2(_, Probe, NotIFragments, _, _),
    member(Fragment, NotIFragments).

is_hybridized_to(Probe, Fragment, 'MluI') :-
    table2(_, Probe, _, MluIFragments, _),
    member(Fragment, MluIFragments).

is_hybridized_to(Probe, Fragment, 'NruI') :-
    table2(_, Probe, _, _, NruIFragments),
    member(Fragment, NruIFragments).
```

where `member(X, Y)` is a built-in predicate which succeeds if `X` is a member of `Y`.

To relieve these difficulties, an object-oriented data representation has been adopted in Lucy. The hybridization relationship between a fragment and probes has been embedded as an attribute of the fragment.

2.2.1 Principle

First of all, we recognize that any kind of datum is an object composed of attributes and represented as a Prolog fact, `object/2`, consisting of a functor, `object`, and two arguments, as follows:

```
object(ObjId, Attributes).
```

where

- `ObjId` is an object identifier which is unique in the entire system and is formed of a *class* and a *local identifier* unique within the class;
- `Attributes` is a set of attributes which constitute the object. The internal representation of attributes is encapsulated in the variable, `Attributes`.

Next, we construct general interface methods which allow retrieval of information from an object without knowing how that object is internally represented as follows:

- `class(ObjId, Class)` returns the class of the object.
- `id(ObjId, LocalId)` returns the local identifier of the object.
- `attribute(ObjId, Attribute)` returns an attribute composed of an attribute name and an attribute value.

2.2.2 Examples

Starting with a restriction fragment, let us consider several objects related with this fragment and see what kinds of information are associated with them. Note that, in this paper, the attributes are represented in the form of a list merely for ease of explanation; a different data structure, more efficient in space and access, is used in the real implementation.

1. **Restriction Fragment:** This defines the 750Kb *NotI* fragment appearing in rows 10 to 13 in Table 1, that has been digested from cell line `WAV-17` with restriction enzyme *NotI*. This fragment was hybridized to four probes: `LA171s`, `SF93`, `512-16P` and `SF13A`. This information was obtaipined in an experiment done by `Denan Wang`, April 1991, and appears in a literature, `Saito et al (1991)`.

```
object('LUCY:fragment'('Denan1991:WAV-17/NotI/750#2'),
       [input_date(1991/4/24),
        digested_from(cell_line('WAV-17')),
        digested_by(restriction_enzyme('NotI')),
        probes([half_linking('LA171s'),clone('SF93'),
                clone('512-16P'),clone('SF13a')]),
        size('Kb'(750)),
        source(ref('Denan Wang (April 1991)')),
        references([ref('Saito et al (1991)')])
       ]).
```

2. **Probe:** One of the probes, `SF93`, was offerred by `Cox`, and registerred in a local clone logbook, Plasmid Book, with a local name, `CLS3048`. It has an *EcoRI* site at one end and a *SalI* site at the other end and is cloned in a `pUC18` vector, and is resistant to `ampicillin`.

```
object('PB:clone'('SF93'),
       [input_date(1991/8/8),
        symbol('SF93'),
        information_source(db('PB/ver.89-11-8')),
        if_confirmed(yes),
        lab_number('CLS:clone'('CLS3048')),
        within([locus('D21S51'),region('21q22')]),
        size('Kb'(2.1)),
        clone_sites([restriction_site('EcoRI'),
                     restriction_site('SalI')]),
        vector(vector('pUC18')),
        vector_size('Kb'(2.7)),
        antibiotic(amp),
        source('PB:contact'('Cox'))
       ]).
```

3. **Locus/Gene:** Clone `SF93` is a representative of locus `D21S51` whose information is found in public database GDB.

```
object('GDB:locus'('D21S51'),
       [input_date(1991/7/5),
       information_source(db('GDB/ver.1.0')),
       sources(['GDB:source'('Korenberg et al (1987)'),
                'GDB:source'('Burmeister et al (1990)')]),
       probes(['GDB:probe'('SF-93')]),
       symbol('D21S51'),
       full_name('DNA Segment, single copy probe SF-93'),
       within([region('21q22.3')]),
       locus_type('DNA'),
       if_cloned(yes),
       assignment_modes(['GDB:assignment_mode'('N'),
                         'GDB:assignment_mode'('S')]),
       certainty(confirmed),
       report(include),
       create_date('Apr 17 1990  1:20:46:000AM'),
       modify_date('Nov 25 1990  2:01:29:460PM'),
       approved_date('Sep  8 1990 11:06:13:320PM')]).
```

4. **Contact:** The person simply referred to as `Cox` in the Plasmid Book is `David R. Cox` whose detailed information is found also in public database GDB.

```
object('PB:contact'('Cox'), Attributes) :-
      object(contact('David R. Cox'), Attributes).

object('GDB:contact'('David R. Cox'),
      [input_date(1991/7/5),
       information_source(db('GDB/ver.1.0')),
       'GDB:idx'('GDB:contact'(1148)),
       symbol('David R. Cox'),
       contact_address(['Univ. of California at San Francisco',
                        'Dept. of Pediatrics/Psych/Biochem',
                        '505 Parnassus Ave., Box 0106']),
       city_address('San Francisco'),
       state_address('CA'),
       post_code('94143'),
       country_name('USA'),
       email_address('rjb@canctr.mc.duke.edu'),
       phone_number('1-(415) 476-4212'),
       'FAX_number'('1-(415) 476-9843')
      ]).
```

5. **Literature:** The mapping effort concerning the above restriction fragment and clones was presented in the literature, `Saito et al (1991)`, as follows:

```
object('LUCY:reference'('Saito et al (1991)'),
      [input_date(1991/4/24),
       kind(paper),
       authors(['Akihiko Saito', 'Jose P. Abado',
                'Denan Wang', 'Misao Ohki',
                'Charles R. Cantor', 'Cassandra L. Smith']),
       title('Construction and Characterization of a NotI
              Linking Library of Human Chromosome 21'),
       journal('Genomics'),
       volume(10),
       year(1991)
      ]).
```

Thus, not only biological data but also personal information and literature references are all represented in an object-oriented manner.

2.2.3 Restricting Classifications

Many biological terms have been introduced so far, such as chromosome, locus, gene, probe, clone and restriction fragment, but each of them represents *just a piece of DNA*. For example, when a restriction fragment is cloned, it is called a clone. When it is used for hybridization and gives information as a landmark, it is called a probe. The more biological experiments are applied to an object, the more names and attributes are given to it. Also a set of constraints over attributes forms a new category (or class). For example, when a clone is sequenced and found to contain some restriction site in it, it is called a linking clone; if the restriction site is an *NotI*, then it is called an *NotI* linking clone.

```
object(linking_clone(Id), Attributes) :-
      object(clone(Id), Attributes),
      find_attribute(Attributes, categories(Categories)),
      is_member(linking_clone, Categories).

object('NotI_linking_clone'(Id), Attributes) :-
      object(linking(Id), Attributes),
      find_attribute(Attributes,
                  linking_site(restriction_enzyme('NotI'))).
```

As a principle, objects may have no class when they are created. Classification is made as more attributes are accumulated and properties are found through later experiements.

2.2.4 Broading Classifications

The information sources constituting Lucy cover more than forty maps collected from publications and several different kinds of public and local biological databases, as shown in Figure 3.

In general, in integrating databases, mainly two kinds of strategies are considered: (1) one is to distill the source databases and unite them into a single database, and (2) the other is to preserve the original form of the source database and provide a bridging interface over them.

Similar biological experiments are being done in parallel at different places. As a result, similar data are accumulated in different databases or even in a single database independently. Also, each datum stored in a version of a database might be corrected or changed through later experiments and reported in a later version of the database. In integrating a genome database, preserving the redundancy and inconsistency of data is a substantial effort.

As a result, the second integration strategy taken in Lucy keeps track of the redundancy and inconsistency. The following program provides a bridging interface to bundle clones which are stored in various sources. Any clone can be referred to with a class name, `clone`, while

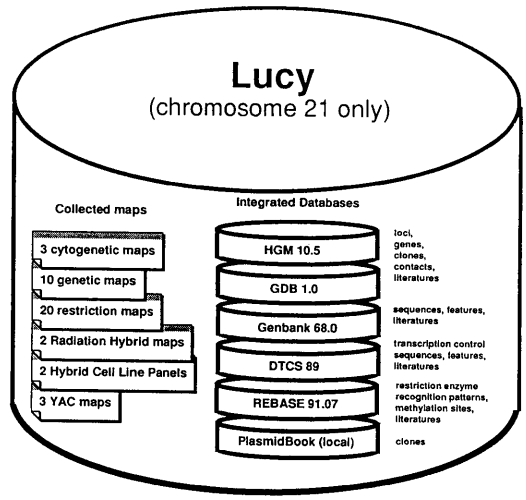


Figure 3: Information sources on chromosome 21, integrated in Lucy

their original class name is preceeded with the database name, such as `PB:clone`. It preserves its origin as an additional attribute, `self(Obj)`

```
object(clone(Id), Attributes) :-
      member(Class, ['GDB:probe', 'PB:clone', 'CLS:clone',
                     'LA:clone', 'LL:clone', 'YZ:clone',
                     'Sakaki:clone', 'LUCY:clone']),
      object_id(Obj, Class, Id),
      object(Obj, Attributes0),
      add_attribute(Attributes0, self(Obj), Attributes).
      ).
```

In summary, the notion of class introduced in Lucy is *loose* unlike such a stringent notion as "class-as-template" which is widely adopted in object-oriented programming languages [41, 78, 91].

3 Constructing a Global Map from Fragmentary Maps

In order to understand mapping information in a visual form, a general graphic interface, GenoGraphics [95, 43], has been hooked up to the Lucy database system.

As shown in Figure 3, those maps collected in Lucy have a variety of range and scaling unit. Some maps cover q-telomeric regions, some do centromeric regions, and many others do some specific region (or island) such as locus D21S13 that is concerned with the Alzheimer disease. Also physical maps are measured their coordinates in Kb (kilo base), genetic maps are in cM (centi-Morgans), and cytognetic maps are in ratio (%).

For the moment, even the total genome size of chromosome 21 is not precisely determined. If every object in maps measured in percentage were specified with an

absolute coordinate, the coordinate would have to be modified every time the total genome size is corrected through later experiments. Similarly, the exact position of the D21S13 locus is not fixed, either. Every time a more precise position were determined for locus D21S13, the coordinates of all maps around the locus would have to be changed.

3.1 Map Expression

First of all, objects in each fragmentary map should be addressed in a local coordinate system within the map, so that the specification of coordinates of objects does not need to be modified in the event that their island floats around. Namely, a relative addressing coordinate system is required. Next, for those fragmentary maps associated with some landmark, when the landmark moves around, they should follow without modification in their coordinate system.

In Lucy, a map representation language called an *map expression* has been introduced, which allows a map to be represented in a local coordinate system and in a relative addressing manner, and to be linked to another with an anchoring mechanism. The syntax of a map expression is defined as follows:

```
<MapExp> ::=   <Obj>
             | ':=' <MapExp>          | <MapExp> '=:'
             | <MapExp> ':<' <MapExp> | <MapExp> '<:' <MapExp>
             | '[' <MapExp> , ... , <MapExp> ']'
```

1. **Relative Addressing:** Two notions are associated with a map expression: one is the *current position* and the other is the *current direction.*

   (a) **Linear-Ordering**
   Expressions `A :< B` and `A <: B` mean, in common, that `A` is left of `B`; additionaly, the former means that `B` is evaluated after `A` is done, while the latter does that `A` is evaluated after `B` is done.

   (b) **Changing the Current Evaluation Direction**
   Expression `:= A` means to put the left bound of `A` at the current position and proceed the evaluation rightward, while expressin `A =:` means to put the right bound of `A` at the current position and proceed the evaluation leftward.

   (c) **Multi-Pinning**
   Expression `[A, B] <: C` means that `A` is left of `C` as well as `B` is left of `C`.

2. **Anchoring:** Objects constituting a map expression include positions and anchors (positions associated with labels). A label is globally accessible beyond a map expression so that it connects one map expression with another.

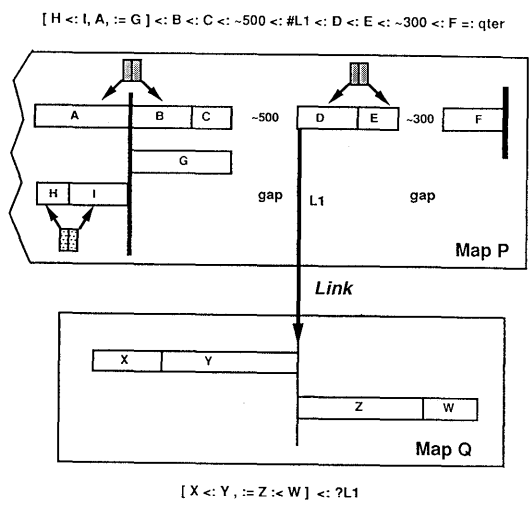


Figure 4: Map expressions

   (a) **Memorizing an Anchor**
   Expression `#L :< B` means to memorize the left bound of `B` under the label `L`.

   (b) **Referring to an Anchor**
   Expression `A <: ?L` means to refer to an anchor labelled with `L` to take it as the right bound of `A`.

Figure 4 illustrates an example of expressing two fragmentary maps, `P` and `Q`, which are linked up at the middle. Map `P` starts with the q-telomere which is followed by fragment `F`, a 300Kb gap, fragment `E`, fragment `D`, a 500Kb gap, fragment `C` and fragment `B`. At the left bound of fragment `B`, three other fragmentary maps start: one map proceeds pinning leftward on fragment `I` and then on fragment `H`, one map goes leftward from fragment `A`, and the other map goes rightward from fragment `G`. The position of the left bound of fragment `D` is labelled `L1` to be an anchor for map `Q`. Map `Q` contains two fragmentary maps starting with the anchor labelled `L1`. One map proceeds pinning with `Y` and then `X` leftward from the anchor, and the other does with `Z` and then `W` rightward from the anchor.

Figures 5, 6 and 7 show those maps represented in map expressions, using GenoGraphics.

Figure 5: this is an *Not*I restriction map around the q-telomere region of chromosome 21, some of whose data have been introduced in Table 1. *Not*I fragments and sites are shown in light green; gray lines denote hybridization signals between fragments and probes. Thus an interpretation of biological data is visualized to help understanding and verifying the mapping process.

Figure 6: in [34], regions are defined based on breakpoints (bounds) of various cell lines. The map of regions

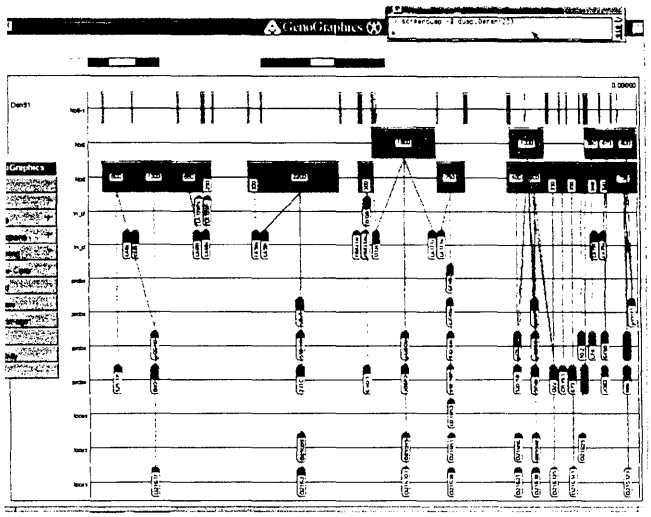


Figure 5: Visualizing a restriction map with hybridization signals

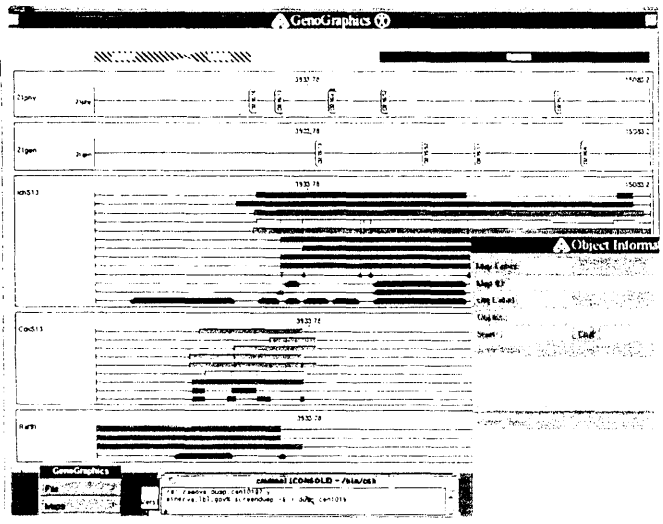


Figure 7: Three maps around locus D21S13

(labelled `Gar90`) is expressed with the breakpoints of the cell line panel (labelled `bkptst`) as anchors. For example, region `A7` is formed with the left bound of cell line `1;21` and the left bound of cell line `ACEM-2`.

Figure 7: three restriction maps, labelled `IchS13`, `CoxS13` and `Raf91`, are those around the `D21S13` locus whose position is given in the chromosome 21 physical anchor map (labelled `21phy`), scaled in percentage.

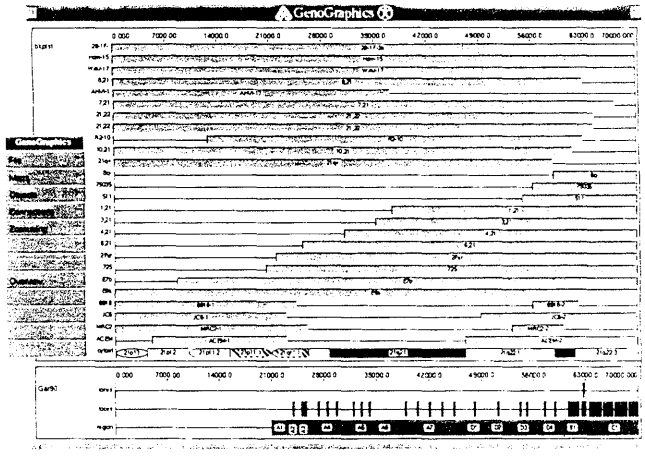


Figure 6: Gardiner's region map generated from a cell line panel

4 Inquiry to Lucy

This section presents the current status of queries Lucy can currently handle.

Concering the map visualized in Figure 5, the mapping effort started with the q-telomere and reached around locus D21S17 where a 1300Kb *Not*I fragment was pinned down. The following queries are issued to retrieve information related with this region so that the mapping effort can be advanced toward the centromere.

1. Regarding locus D21S17, its regional information is retrieved.

```
| ?- get_attributes(locus('D21S17'), [self(S)), within(Rs)]),
     print_object(Rs-S), fail.
------------------------------------------------------------------
Object Id:
    -([region(21q22.1-q22.2)], GDB:locus(D21S17))
------------------------------------------------------------------
Object Id:
    -([region(21q21.2-qter)], HGM10.5:locus(D21S17))
------------------------------------------------------------------
Object Id:
    -([gardiner_region(B1), region(21q22.3)], LUCY:locus(D21S17))
------------------------------------------------------------------
```

The region recorded in GDB is narrower than the one in HGM10.5 which is the predecessor of GDB. Also the D21S17 locus is assigned to region B1 in Gardiner's map shown in Figure 6.

2. Then, objects which occur left of D21S17 in all maps on which D21S17 occurs are retrieved.

```
| ?- setof(Obj-MID, Os^( occurs_on(map(MID), locus('D21S17')),
                         ordered_objects_on_map(map(MID), Os),
                         left_to(Obj, locus('D21S17'), Os) ),
          OMs),
     keymerge(OMs, KOMs), !,
     member(OM, KOMs), print_object(OM), fail.
----------------------------------------------------------------
Object Id:
    -(clone(pGSH8), [chr21_Denan1991_physical_around_21q22.3])
----------------------------------------------------------------
Object Id:
    -(gardiner_region(B1), [chr21_Gardiner1990])
----------------------------------------------------------------
Object Id:
    -(locus(D21S58), [chr21_Burmeister1991_RH, chr21_Petersen1991
    _female_meiosis, chr21_Petersen1991_male_meiosis, chr21_Tanzi
    1988_female, chr21_Tanzi1988_male, chr21_Tanzi1988_sex_averag
    ed, chr21_physical_anchors])
----------------------------------------------------------------
Object Id:
    -(locus(D21S82), [chr21_Warren1989_female_meiosis, chr21_Warr
    en1989_male_meiosis])
----------------------------------------------------------------
```

Beside clone pGSH8 and region B1, loci D21S58 and D21S82 are reported.

3. For the D21S58 locus, its regional information is retrieved.

```
| ?- get_attributes(locus('D21S58'), [self(S), within(Rs)]),
     print_object(Rs-S), fail.
----------------------------------------------------------------
Object Id:
    -([region(21q22.1-q22.2)], GDB:locus(D21S58))
----------------------------------------------------------------
Object Id:
    -([region(21q21)], HGM10.5:locus(D21S58))
----------------------------------------------------------------
Object Id:
    -([gardiner_region(D4)], LUCY:locus(D21S58))
----------------------------------------------------------------
```

Although the answers from GDB and HGM10.5 conflict, the locus is assigned to region D4 in Gardiner's map, which is to the left of region B1.

4. In order to grasp what more loci reside further left, all loci not only in region D4 but also in every D region are retrieved.

```
| ?- setof(R-Id, Rs^( get_attribute(locus(Id), within(Rs)),
                      member(gardiner_region(R), Rs),
                      substring(R, "D") ),
          RIs),
     keymerge(RIs, KRIs), !,
     member(KRI, KRIs), print_object(KRI), fail.
----------------------------------------------------------------
Object Id:
    -(D1, [D21S54])
----------------------------------------------------------------
Object Id:
    -(D2, [D21S93])
----------------------------------------------------------------
Object Id:
    -(D3, [D21S63, SOD1])
----------------------------------------------------------------
Object Id:
    -(D4, [D21S58, D21S65])
----------------------------------------------------------------
```

5. Finally, detailed information about locus D21S58 is retrived.

```
— ?- print object(locus('D21S58')).
----------------------------------------------------------------
Categories:
   [1] locus

Input Date:
   1991/8/5

Investigators:
   [1] contact(P. C. Watkins)

Object Id:
   locus(D21S58)

Probes:
   [1] clone(524-5P)

References:
   [1] Katheleen Gardiner, Michel Horisberger, Jan Kraus, Umadev
   i Tantravahi, Julie Korenberg, Veena Rao, Shyam Reddy, David
   Patterson, "Analysis of human chromosome 21: correlation of p
   hysical and cytogenetic maps; gene and CpG island distributio
   ns", The EMBO Jounal, 9, 25-34, 1990

   [2] Michael B. Petersen, Susan A. Slaugenhaupt, John G. Lewis
   , Andrew C. Warren, Aravinda Chakravarti, Stylianos Antonarak
   is, "A Genetic Linkage Map of 27 Markers on Human Chromosom 2
   1", Genomics, 9, 407-419, 1991

Self:
   LUCY:locus(D21S58)

Within:
   [1] gardiner region(D4)

[GDB/ver.1.0] Approved date:
   Sep  8 1990 10:57:11:140PM

[GDB/ver.1.0] Assignment modes:
   [1] somatic cell hybrids

[GDB/ver.1.0] Certainty:
   confirmed

[GDB/ver.1.0] Create date:
   Jun 18 1989  9:42:08:000AM

[GDB/ver.1.0] Full name:
   DNA Segment, single copy probe pPW524-5P

[GDB/ver.1.0] GDB:idx:
   GDB:locus(8242)

[GDB/ver.1.0] If cloned:
   yes

[GDB/ver.1.0] Information source:
   db(GDB/ver.1.0)

[GDB/ver.1.0] Input Date:
   1991/7/5

[GDB/ver.1.0] Locus type:
   DNA

[GDB/ver.1.0] Modify date:
   Nov 25 1990  2:01:47:640PM

[GDB/ver.1.0] Polymorphism type:
   polymorphic

[GDB/ver.1.0] Probes:
   [1] GDB:probe(pPW524-5P)

[GDB/ver.1.0] Report:
   include

[GDB/ver.1.0] Self:
   GDB:locus(D21S58)

[GDB/ver.1.0] Sources:
   [1] P. C. Watkins, R. E. Tanzi, J. Roy, N. Stuart, P. Stanisl
   ovitis, J. F. Gusella, "A cosmid clone genetic linkage map of
    chromosome 21 and localization of the breast cancer estrogen
   -inducible (BCEI) gene.", Cytogenet Cell Genet, 46, 712, 1987

   [2] M. Van Keuren, H. Drabkin, P. Watkins, J. Gusella, D. Pat
   terson, "Regional mapping of DNA sequences to chromosome 21."
   , Cytogenet Cell Genet, 40, 768-769, 1985

   [3] P. C. Watkins, P. A. Watkins, N. Hoffman, P. Stanisloviti
   s, "Isolation of single-copy probes detecting DNA polymorphis
   ms from a cosmid library of chromosome 21.", Cytogenet Cell G
   enet, 40, 773-774, 1985

   [4] M. L. Van Keuren, P. C. Watkins, H. A. Drabkin, E. W. Jab
   s, J. F. Gusella, D. Patterson, "Regional localization of DNA
    sequences on chromosome 21 using somatic cell hybrids.", Am
   J Hum Genet, 38, 793-804, Jun 1986

   [5] M. Burmeister, S. Kim, R. Price, T. de Lange, U. Tantrava
   hi, R. M. Myers, D. R. Cox, "A map of the long arm of human c
   hromosome 21 constructed by radiation hybrid mapping and puls
   ed-field gel electrophoresis", Genomics, In Press, ??, 1990

[GDB/ver.1.0] Symbol:
   D21S58

[GDB/ver.1.0] Within:
   [1] region(21q22.1-q22.2)
```

```
[HGM10.5] # of copies:
   single

[HGM10.5] Assignment modes:
   [1] somatic cell hybrids

[HGM10.5] Categories:
   [1] locus

[HGM10.5] Certainty:
   provisional

[HGM10.5] Information source:
   db(HGM10.5)

[HGM10.5] Input Date:
   1991/3/27

[HGM10.5] Probes:
   [1] clone(pPW524-5P)

[HGM10.5] References:
   [1] ref(Watkins et al (HGM8))

   [2] P. C. Watkins, R. E. Tanzi, K. T. Gibbons, J. V. Tricoli,
    G. Landes, R. Eddy, T. B. Shows, J. F. Gusella, "Isolation o
   f polymorphic DNA segments from human chromosome 21.", Nuclei
   c Acids Res, 13, 6075-88, Sep 1985

   [3] M. L. Van Keuren, P. C. Watkins, H. A. Drabkin, E. W. Jab
   s, J. F. Gusella, D. Patterson, "Regional localization of DNA
    sequences on chromosome 21 using somatic cell hybrids.", Am
   J Hum Genet, 38, 793-804, Jun 1986

   [4] ref(Nakai et al (HGM9))

[HGM10.5] Self:
   HGM10.5:locus(D21S58)

[HGM10.5] Within:
   [1] region(21q21)

-------------------------------
```

Information are reported from publications, GDB and HGM10.5 in that order.

5 Concluding Remarks

Promoted by requirements in various application areas as well as in biology, steady progress in database technology has been made in the last few years [82].

Since the normal-form (1NF or *flat*) relational model [24] was proposed, practice over the years has pointed out its inefficiency in data access and its verbosity in inquiry [25, 28]. The source of both problems is the primitive data structure, the flat relation. In genome databases implemented upon normal form relational database systems, these problems are cast in relief, since the volume and variety of involved data are large and growing. In fact, the number of tables constituting a genome mapping database is apt to be quite large (e.g., 68 tables in LLNL Genome Database [4] and over 100 tables in GDB).

The present work could be regarded as one of the first to have successfully integrated public and local genome databases. The success greatly reflects the application of an object-oriented data representation and logic programming features, which should be the preliminary steps toward object-oriented databases [5, 36, 3, 7, 33, 85, 19] and deductive databases respectively. Through an experience with Lucy, it should be reasonable to conclude that these database technologies will contribute to the development and practice of genome databases.

Object-Oriented Database Technology. Since the notion of object-orientation [41] was invented in the field of programming languages, it has been widely disseminated over the past ten years [78, 91]. The heart of object-orientation, that is encapsulating the internal details of an object, is important for the implementation and retrieval of various kinds of data involved in genome databases. Lucy has only adopted an object-oriented data representation. Other ramparts have not been constructed yet: neither object-specific methods nor class inheritance. They will be future work.

Since the object-orientation was introduced to Lucy, some cases have been found where the framework does not fit naturally but where a nested (NF^2: non-first normal form) relational model [38, 1, 76] would. Here is an example. Given a table of linking clones, an entry for LA171 has been represented as follows:

name	region	# of occurrences MluI	BssHII	SacII	cloned fragments large	small
LA171	21q22.3	1	2	3	3.0	2.1
LA179	21cen	0	3	2	1.1	0.96
:	:	:	:	:	:	:

```
object('LA:clone'('LA171'),
      [input_date(1991/2/11),
       categories([linking_clone]),
       within([region('21q22.3')]),
       cloning_vector(lambda),
       linking_site(restriction_enzyme('NotI')),
       digested_from(genomic_DNA(human)),
       digested_by([restiction_enzyme('EcoRI')]),
       contains([times(restriction_enzyme('MluI'), 1),
                 times(restriction_enzyme('BssHII'), 2),
                 times(restriction_enzyme('SacII'), 3) ]),
       parts(['LA:clone'('LA1711'), 'LA:clone'('LA171s')]),
       references([ref('Saito et el (1991)')])
      ]).

object('LA:clone'('LA1711'),
      [input_date(1991/3/30),
       categories([half_linking_clone]),
       linking_site(restriction_enzyme('NotI')),
       size('Kb'(3.0))
      ]).

object('LA:clone'('LA171s'),
      [input_date(1991/3/30),
       categories([half_linking_clone]),
       size('Kb'(2.1))
      ]).
```

As shown in Table 1, `LA1711` and `LA171s` are those half linking clones which hybridized to fragments, 1800Kb and 750Kb, respectively. When these half linking clones were identified as objects, their sizes, 3.0Kb and 2.1Kb, were encapsulated in these objects. In contrast, consider the number of occurrences of restriction sites. It is questionable that an object should be created for the number of occurrences, such as once, twice or three times. Being part of an attribute, `contains`, the occurrences are stored as a nested relation of the form, `times/2`. For the third and forth columns in the example above, their relational structures are similar, but the meanings of their data imply different implementations. Further studies will be necessary to clarify this problem.

Deductive Database Technology. The necessity of loading an inference mechanism into a database system has been claimed in knowledge-intensive applications [16, 56, 63, 84].

Because most biological knowledge is symbolic rules on the four characters of DNA, there is a potential requirement for rule processing capability. A couple of genome database systems are being developed abreast of Lucy, exploiting logic programming facilities [42, 73, 6, 45]. In Lucy, the inference capability is being used mainly for query management. Few pieces of biological rules have been implemented.

References

[1] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp.191-200, Waterloo, April 1984.

[2] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the 1989 ACM-SIGMOD Conference on Management of Data*, Portland, OR, May 1989.

[3] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of the 1989 ACM-SIGMOD Conference on Management of Data*, Portland, OR, May 1989.

[4] L. K. Ashworth, T. Slezak, M. Yeh, E. Branscomb and A. V. Carrano. Making the LLNL Genome Database 'Biologist Friendly'. *DOE Human Genome Program Contractor-Grantee Workshop*, Santa Fe, NM, February 17-20 1991.

[5] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, 1989.

[6] A. Baehr, G. Dunham, A. Ginsburg, R. Hagstrom, D. Joerg, T. Kazic, H. Matsuda, G. Michaels, R. Overbeek, K. Rudd, C. Smith, R. Taylor, K. Yoshida and D. Zawada. *An Integrated Database to Support Research on Esherichia coli.* Technical Report, Argonne National Laboratory, October 1991.

[7] J. Banerjee, H.-T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou and H. J. Kim. Data model issues for object-oriented applications. *ACM TOIS*, Jan 1987.

[8] F. Bancilhon and S. N. Khoshafian. A calculus of complex objects. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.53-59, March 1986.

[9] Bart Barrell. DNA sequencing: present limitations and prospects for the future. *FASEB Journal*, 5:40-45, 1991.

[10] B. J. Billings, C. L. Smith and C. R. Cantor. New techniques for physical mapping of the human genome. *FASEB Journal*, 5:28-34 (1991).

[11] D. G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1:3-46, 1977.

[12] R. J. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171-216, April-June 1985.

[13] R. J. Brachman, A. Borgida, D. L. McGuinness, and L. A. Resnick. The CLASSIC knowledge representation system, or, KL-ONE: The next generation. *Workshop on Formal Aspects of Semantic Networks*, Santa Catalina Island, CA, February 1989.

[14] E. Branscomb, T. Slezak, R. Pae, D. Galas, A. V. Carrano and M. Waterman. Optimizing restriction fragment fingerprinting methods for ordering large genomic libraries. *Genomics*, 8: 351-66 (1990).

[15] Ivan Bratko. *Prolog: programming for artificial interlligence.* Second Edition, Addison Wesley, 1990.

[16] Bruce G. Buchanan and Edward H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Reading, Addison-Wesley, 1984.

[17] Margit Burmeister, Suwon Kim, E. Roydon Price, Titia de Lange, Umadevi Tantravahi, Richard M. Myers, and David R. Cox. A Map of the Distal Regin of the Long Arm of Human Chromosome 21 Constructed by Radiation Hybrid Mapping and Pulsed-Field Gel Electophoresis. *Genomics*, 9:19-30, 1991.

[18] Paul Butterworth, Allen Otis and Jacob Stein. The GemStone object database management system. *Communication of the ACM*, Vol.34, No.10, pp.64-77, October 1991.

[19] Michael J. Carey, David J. DeWitt and Scott L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data*, Chicago, IL, June 1988.

[20] A. V. Carrano. Estalishing the order of human chromosome-specific DNA fragments. *Basic Life Science*, 46: 37-49 (1988).

[21] A. V. Carrano, J. Lamerdin, L. K. Ashworth, B. Watkins, E. Branscomb, T. Slezak, M. Raff, P. J. de Jong, D. Keith, L. McBride. A high-resolution, fluorescence-based semiautomated method for DNA fingerprinting. *Genomics*, 4: 129-36 (1989).

[22] A. V. Carrano, P. J. de Jong, E. Branscomb, T. Slezak and B. W. Watkins. Constructing chromosome- and region-specific cosmid maps of the human genome. *Genome*, 31: 1059-65, (1989).

[23] M. J. Cinkosky and J. W. Fickett. SIGMA: Software for Integrated Genome Map Assembly. *Proc. of the 11th International Human Gene Mapping Workshop (HGM11)*, London, August 18-22, 1991.

[24] E. F. Codd, A Relational Model of Data for Large Shared Data Banks. *Communication of ACM*, 13:6, 1970.

[25] E. F. Codd. *The Relational Model for Database Management, Version 2.* Addison Wesley, 1990.

[26] A. Coulson, J. Sulston, S. Brenner and J. Karn. Toward a physical Map of the genome of the nematode, Caenorhabditis elegans. *Proc. Natl. Acad. ci. USA*, 83: 7821-7825, 1986.

[27] David R. Cox, Margit Burmeister, E. Roydon Price, Suwan Kim, Richard M. Myers. Radiation Hybrid Mapping: A Somatic Cell Genetic Method for Constructing High-Resolution of Mammalian Chromosomes. *Science*, 250: 245–250, 1990.

[28] C. J. Date. *An Introduction to Database Systems, Volume I*, Fifth Edition, Reading, Addison Wesley, 1990.

[29] Kay E. Davies and Shirley M. Tilghman, editors. *Genome Analysis Volume 1: Genetic and Physical Mapping.* Cold Spring Harbor Laboratory Press, 1990.

[30] *Human Genome 1989-90 Program Report.* DOE/ER-0446P, U.S. Department of Energy, March 1990.

[31] *Understanding Our Genetic Inheritance, The U.S. Human Genome Project: The First Five Years FY 1991-1995.* DOE/ER-0452P, U.S. Department of Energy.

[32] R. Durbin, S. Dear, T. Gleeson, P. Green, L. Hillier, C. Lee, R. Staden and J. Thierry-Mieg. Software for the C. Elegans Genome Project. *Genome Mapping and Sequencing Meeting*, Cold Spring Harbor Laboratory, NY, May 8-12 1991.

[33] D. Fishman et al. Iris: An object-oriented database management system. *ACM TOIS*, Vol.5, No.1, pp.48-69, January 1986.

[34] Katheleen Gardiner, Michel Hoisberger, Jan Kraus, Umadevi Tantravahi, Julie Korenberg, Venna Rao, Shyam Reddy and David Patterson. Analysis of human chromosome 21: coorrelation of physical and cytogenetic maps; gene and CpG island distributions. *The EMBO Journal*, vol.9, no.1, pp.25-34, 1990.

[35] D. Garza, J. W. Ajioka, D. T. Burke and D. L. Hartl. Mapping the Drosophila genome with yeast artificial chromosomes. *Science*, 246: 641-6 (1989).

[36] O. Deux et al. The O_2 system. *Communication of the ACM*, Vol.34, No.10, pp.34-48, October 1991.

[37] J. W. Fickett, M. J. Cinkosky, D. Sorensen and C. Burks. Integrated Maps: A Model and Supporting Tools. *Proc. of the 11th International Human Gene Mapping Workshop (HGM11)*, London, August 18-22, 1991.

[38] P. Fisher and S. Thomas. Operators for non-first-normal-form relations. In *Proceedings of the 7th International Computer Software Applications Conference*, Chicago, November 1983.

[39] Karen A. Frenkel. The Human Genome Project and Informatics. *Communication of ACM*, Vol.34, No.11, 40–51, 1991.

[40] *GDB and OMIM Quick Guide (Version 4.0).* GDB/OMIM User Support, The William H. Welch Medical Library, Baltimore, July 1991.

[41] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[42] P. M. D. Gray and R. J. Lucas, editors. *Prolog and Databases: implementations and new directions.* Ellis Horwood, series in Artificial Intelligence, 1988.

[43] Ray Hagstrom. GenoGraphics. *International Chromosome 21 Workshop*, Denver, CO, April 10-11, 1991.

[44] L. Hood, R. Kaiser, B. Koop and T. Hunkapiller. Large-Scale DNA Sequencing. *DOE Human Genome Program Contractor-Grantee Workshop*, Santa Fe, NM, February 17-20 1991.

[45] Toni Kazic and Shalom Tsur. Building a Metabolic Function Knowledgebase System. *Workshop on Biological Applications in Logic Programming, ILPS'92*, San Diego, CA, November 1991.

[46] Y. Kohara, K. Akiyama and K. Isono. The Physical Map of the Whole E.coli Chromosome: Application of a New Strategy for Rapid Analysis and Sorting of a Lrge Genomic Library. *Cell*, 50: 495-508, 1987.

[47] M. Kosowsky, C. Blake, D. Bradt, J. Eppig, P. Grant, L. Mobraaten, J. Nadeau, J. Ormsby, A. Reiner, S. Rockwood, J. Saffer, T. Snell and M. Vollmer. Integration and Graphical Display of Genomic Data: *The Encycolopedia of the Mouse Genome. Genome Mapping and Sequencing Meeting*, Cold Spring Harbor Laboratory, NY, May 8-12 1991.

[48] Donald Johanson and Maitland Edey. *Lucy: The Beginnings of Human Kind.* Simon & Schuster, 1981.

[49] Peter Karp. *A Knowledge Base of the Chemical Compounds of Intermediary Metabolism.* unpublished, SRI International, September 1991.

[50] G. Kiernan, C. de Maindreville and E. Simon. Making Deductive Database a Practical Technology: a step forward. In *Proceedings of the 1990 ACM-SIGMOD Conference on Management of Data*, Atlantic City, NJ, May 1990.

[51] Charles Lamb, Gordon Landis, Jack Orenstein, Dan Weinreb. The ObjectStore database system. *Communication of the ACM*, Vol.34, No.10, pp.50-63, October 1991.

[52] Eric S. Lander and Michael S. Waterman. Genomic Mapping by Fingerprinting Random Clones: A Methematical Analysis. *Genomics*, 2:231-239, 1988.

[53] Eric S. Lander, Robert Langridge and Damian M. Saccocio. Mapping and Interpreting Biological Information. *Communication of ACM*, Vol.34, No.11, 32–39, 1991.

[54] Hans Lehrach, Radoje Drmanac, Jorg Hoheisel, Zoia Larin, Greg Lennon, Anthony P. Monaco, Dean Nizetic, Gunther Zehetner and Annemarie Poustka. Hybridization Fingerprinting in Genome Mapping and Sequencing. *Genome Analysis Volume 1: Genetic and Physical Mapping*, pp.39-81, Cold Spring Harbor Laboratory Press, 1990.

[55] A. J. Link and M. V. Olson. Physical map of the Saccaromyces cerevisia genome at 110-kilobase resolution. *Genetics*, 127: 681-98 (1991).

[56] Douglas B. Lenat and R. V. Guha. The Evolution of CycL, The Cyc Representation Language. *SIGART Bulletin*, Vol. 2, No. 3, ACM Press, June 1991.

[57] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Addison-Wesley, 1990.

[58] S. Lewis, W. Johnston, V. Markowitz, J. McCarthy, F. Olken and M. Zorn. The Chromosome Inforamtion System. *DOE Human Genome Program Contractor-Grantee Workshop*, Santa Fe, NM, February 17-20 1991.

[59] Y. Lien. Hierarchical schemata for relational databases. *ACM Transactions on Database Systems*, Vol.6, No.1, pp.48-69, March 1981.

[60] David Lipman and James Ostel. Entrez: Sequences. In *Proceedings of the 11th International Human Gene Mapping Workshop (HGM11)*, London, August 18-22, 1991.

[61] Joseph Locker and Gregory Buzard. A dictionary of transcription control sequences. *DNA Sequence - Journal of DNA Sequencing and Mapping*, Vol.1, pp.3-11, 1990.

[62] Lohman et al. Extensions to Starburst: objects, types, functions and rules. *Communication of the ACM*, Vol.34, No.10, pp.94-109, October 1991.

[63] D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of ACM SIGMOD 89*, pp.215-224, Portland OR, May 1989.

[64] A. Makinouchi. A consideration on normal form of not-neceissarily-normalized relation in the relational data model. In *Proceedings of the Third International Conference on Very Large Databases*, pp.447-453, Tokyo, October 1977.

[65] Victor A. Mckusick. Current trends in mapping human genes. *FASEB Journal*, 5: 12-20, 1991.

[66] H. W. Mohrenweiser, K. M. Tynan, E. W. Branscomb, P. J. de Jong, A. Olsen, B. Trask and A. V. Carrano. Development of an integrated genetic functional physical map of human chromosome 19. In *Proceedings of the 11th International Human Gene Mapping Workshop (HGM11)*, London, August 18-22, 1991.

[67] Fumio Mizoguchi. Knowledge Representation and Knowlegdge Programming. (in Japanese) *Computer Software*, Vol. 8, No. 4, JSSS, July 1991.

[68] D. Nelson and J. W. Fickett. An Electronic Laboratory Notebook for Data Management in Physical Mapping. *DOE Human Genome Program Contractor-Grantee Workshop*, Santa Fe, NM, February 17-20 1991.

[69] M. V. Olson, J. E. Dutchik, M. Y. Graham, G. M. Brodeur, C. Helms, M. Frank, M. MacCollin, R. Scheinman and T. Frand. Random-clone strategy for genomic restriction mapping in yeast. *Proc. Natl. Acad. Sci. USA*, 83: 7826-7830, 1986.

[70] Marnard Olson, Leroy Hood, Charles Cantor and David Botstein. A Common Language for Physical Mapping of the Human Genome. *Science*, 245: 28-29, September 1989.

[71] James Ostel. *GenInfo Backbone Database Overview (Version 1.0)*. Technical Report, NLM, NIH, Bethesda, MD, June 1990.

[72] *GenInfo ASN.1 Syntax: Sequences (Version 0.50)*. Technical Report, NCBI, NLM, NIH, Bethesda, MD, National Institutes of Health, MD, 1991.

[73] Norman W. Paton and Peter M. D. Gray. An object-oriented database for strage and analysis of protein structure data. In Reading [42].

[74] Mark L. Pearson and Dieter Söll, The Human Genome Project: a paradigm for information management in the life sciences. *FASEB Journal*, 5: 35-39, 1991.

[75] Peter Pearson. The Genome Data Base. *Proc. of the 11th International Human Gene Mapping Workshop (HGM11)*, London, August 18-22, 1991.

[76] Mark A. Roth and Henry F. Korth. The Design of ¬1NF Relational Databases into Nested Normal Forma. In *Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data*, pp.143-159, San Francisco, May 1987.

[77] Akihiko Saito, Jose P. Abado, Denan Wang, Misao Ohki, Charles R. Cantor and Cassandra L. Smith. Construction and Characterization of a NotI Linking Lbrary of Human Chromosome 21. *Genomics*, 10, 1991.

[78] John H. Saunders. A Survey of Object-Oriented Programming Languages. In *Journal of Object-Oriented Programming*, Vol.1, No.6, SIGS Publications, March/April 1989.

[79] H. Scheck and P. Postor. Data structures for an integrated data base management and information retrieval system. In *Proceedings of the Eighth International Conference on Very Large Datbases*, pp.197-207, Mexico City, September 1982.

[80] "Genome Databases." *Science*, 254: 201-7 (1991).

[81] David B. Searls. *The Computational Linguistics of Biological Sequences*. Technical Report CAIT-KSA-9010, Center for Advanced Information Technology, UNISYS, PA, September 1990.

[82] Avi Silberschatz, Michael Stonebraker, Jeff Ullman (editors of the specicial issues on next-generation database systems). Database Systems: Achievements and Opportunities. *Communication of the ACM*, Vol.34, No.10, pp.110-120, October 1991.

[83] J. C. Stephens, M. L. Cavanaugh, M. I. Gradie, M. L. Mador and K. K. Kidd. Mapping the human genome: current status. *Science*, 250: 237-44 (1990).

[84] Michael Stonebraker et al. On rules, procedures, caching and views. In *Proceedings of the 1990 ACM-SIGMOD Conference on Management of Data*, Atlantic City, NJ, June 1990.

[85] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communication of the ACM*, Vol.34, No.10, pp.78-92, October 1991.

[86] J. Claiborne Stephens, Mark L. Cavanaugh, Margaret I. Gradie, Martin L. Mador and Kenneth K. Kidd. Mapping the Human Genome: Current Status. *Science*, 250: 237-250, 1991.

[87] Shunichi Uchida and Kaoru Yoshida. The Fifth Generation Computer Technology and Biological Sequencing. In *Proceedings of Workshop on Advanced Computer Technologies and Biological Sequencing*, Argonne National Laboratory, pp.28-36, November 1988.

[88] Denan Wang, Hong Fang, Charles R. Cantor and Cassandra L. Smith. A Contiguous NotI Restriction Map of Band q22.3 of Human Chromosome 21. to appear in *Proceedings of National Accademy of Science U.S.A.*, 1992.

[89] James Dewey Watson and Robert Mullan Cook-Deegan. Origins of the human genome project. *FASEB Journal*, 5: 8-11, 1991.

[90] James D. Watson, John Tooze, and David T. Kurtz. *Recombinant DNA – A Short Course –*. Scintific American Books, NY,1983.

[91] Kaoru Yoshida. *A'UM: A Stream-Based Concurrent Object-Oriented Programming Lanugage*. Ph.D thesis, Keio University, Japan, March 1990.

[92] Kaoru Yoshida, Cassandra L. Smith, Charles R. Cantor and Ross Overbeek, How will logic programming benefit genome analysis? *DOE Human Genome Program Contractor-Grantee Workshop*, Santa Fe, NM, February 17-20 1991.

[93] Kaoru Yoshida, Ross Overbeek, David Zawada, Charles R. Cantor and Cassandra L. Smith. Prototyping a Mapping Database of Chromosome 21. *Genome Mapping and Sequencing Meeting*, Cold Spring Harbor Laboratory, NY, May 8-12 1991.

[94] Kaoru Yoshida and Cassandra Smith. Key Features in Building a Physical Mapping Database System – Through an Experience of Developing a Human Chromosome 21 Mapping Knowledge Base System –. *The International Conference on the Human Genome (Human Genome III)*, San Diego, CA, October 21-23, 1991.

[95] David Zawada. *GenoGraphics for OpenWindows – v1.1 alpha*. Technical Report, Argonne National Laboratory, August 1991. (GenoGraphics is available via anonymous FTP from info.mcs.anl.gov (140.332.20.2),

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Integrated System for Protein Information Processing

Hidetoshi Tanaka

Institute for New Generation Computer Technology (ICOT)
1-4-28, Mita, Minato-ku, Tokyo 108 Japan
htanaka@icot.or.jp

Abstract

This paper describes requirements for databases and DBMS in protein information processing, and an experiment on a privately integrated protein knowledge base. We consider an integrated DBMS-KRL system for an integrated protein KB-DB.

In order to clarify unknown functions of proteins via empirical approaches, existing public databases should be integrated as part of a private knowledge base, which can proceed biological knowledge discovery. And DBMS-KRL system should support representability, parallel processing, information retrieval, advanced query processing, and quality management techniques for protein information.

DBMS Kappa-P and KRL *QUIXOTE*, both designed at ICOT, perform a useful role in processing protein information. Kappa-P is for efficiency by its parallel processing and extensibility, while *QUIXOTE* is for advanced query processing and quality management, by its representational flexibility of object identification and module.

1 Introduction

Molecular biological information processing is increasing in importance, as biological laboratories are improving in their computational environments and biological data are augmenting far more faster than biologists' understanding. To speed up converting such data into biological knowledge, biological database should help biologists by providing conveniences in storing, browsing, and query processing.

Molecular biological databases have two categories: public databases and private databases. Public databases have hundreds of Mbytes of various data: sequence, structure, functions, and other indispensable auxiliary information of DNA, RNA, and protein.

There are variation in their sizes and data structures. As for the sizes, PIR Release 30 (1991) has the proteins of 1 residue through to 6048 residues. GenBank Release 70 (1991) has the regions (*loci*) of the DNA sequences of 3 bases through to 229354 bases. As for the data structures, for example, the feature descriptions of proteins or loci require multiple nested structures. A protein often consists of plural amino acid sequences. A sequence of *eucaryote* is often coded in several separate DNA regions (*exons*) with separate *expression regulatory region*. The structure of regulatory regions is so unclear that we can only describe them as nested patterns of DNAs.

Public databases are maintained under international cooperation or specific volunteers, and provide most molecular biological data freely. Although the amount of data is increasing rapidly, recent dynamic improvements of machine environments allow biologists to store such data in their own small systems, and to use them as part of value-added private databases.

Such environments also allow them to create a database including their own experimental results, make cross-references between public and private databases, add customized query processing facilities, and try to conduct knowledge discovery by extracting rules from data.

In this paper, we focus on such privately integrated databases which are developed as part of the molecular biological information processing system of the FGCS project.

As an example, we are building an integrated protein knowledge base in the framework of deductive object oriented database (DOOD), which consists of a knowledge representation language (KRL) *QUIXOTE* and a DBMS Kappa-P. The reason why we choose protein information is due to their moderate amount for storage and study. As biological applications are very new, we had to check the appropriateness of the system and request to add several facilities to it.

We have developed Kappa-P and *QUIXOTE* on a parallel inference machine PIM. Kappa-P employs a nested relational model, and has a facility of extensible DBMS, which appears to be suitable for parallel processing and sequence retrieval.

QUIXOTE is based on a concept of DOOD. It provides a capability of advanced query processing, rich concepts such as module, identification, subsumption relation, and flexibility in describing knowledge.

This paper describes two types of system integration. One is database integration of protein information. The other is DB-KB (database and knowledge base) integration in Kappa-P + QUIXOTE. These are shown in Section 6.

We describe the requirements for databases and DBMSs in Section 2. Overviews of the protein databases we are focusing on are described in Section 3. The suitability of Kappa-P and QUIXOTE as ingredients of our integrated knowledge base system is discussed in Sections 4 and 5.

2 Requirements for Biological Database Systems

2.1 Requirements for Databases

Most of the biologists' requirements for existing molecular biological databases are concentrated into the problem: it is difficult to access several databases at once. It is because the differences between the databases: the attributes' meanings, the values' variations, and their relations, must be understood beforehand.

Such requirements are solved by integrating them. There are three approaches to database integration.

Standardization

Standardization is the most fundamental integration. It provides the simplest environment for the wide use of databases.

CODATA (Committee on Data for Science and Technology) in ICSU (International Council of Scientific Unions) proposed standardization of attributes to realize the virtual integrated database [JIPID 90]. The schema of every public database should be a subset of the virtual schema. NLM (National Library of Medicine) provides GenInfo Backbone Database [NCBI 90]. According to [NCBI 90], it is built as a standardized primary database, which is assumed to be a basis for secondary, value-added databases for the specialized interests of different biologists.

Determining a standard, however, is expensive. It is almost impossible to make the widest virtual schema that covers all attributes of protein information. We should accumulate experiences in creating and using most private databases as before. Moreover, both standardizations started so recently that they have not so widely distributed yet.

Integrated User Interface

Making an integrated user interface is the fastest way of getting an integrated environment. It generally provides not only query processing facilities but also visual browsing facilities, which are quite attractive and useful for biologists. It used to need a lot of cost, however, to remake an interface when a new database is to be added.

GeneWorks[1] and Entrez[2] provide integrated environments to enable access to existing DNA / amino acid sequence databases, although they are packaged for browsing only, from a PC or Mac. They are not for adding new applications or new databases.

S. Smith et al. (Harvard U.) are developing an environment for genetic data analysis (GDE[3]) which will help access several databases at once by providing data exchange tools between representative databases. The first version is based on a *multiple alignment editor* and allows tools for sequence analysis to be included in the system. It can reduce efforts for the interface remaking by rich widgets. It has also just started and further improvement is expected.

Integrated Knowledge Base

The integrated knowledge base is our approach. It consists of two stages: to represent all facts in one language, and to supplement the rules necessary to get and use the facts (see Fig. 1). The former corresponds to standardization, and realizes a syntactically integrated database. Not only existing public databases but also private databases are integrated. In order to provide an useful integrated database system, efficient DBMS which allows to store and to access complex data easily.

The latter stage converts a database into a knowledge base, by accumulating supplementary knowledge, which are rules or facts recognized by biologists themselves. It seems almost impossible to define common operations to all knowledge just as relational algebra to relational database. Thus, new concepts had better be introduced to the mechanism, so that each (or a cluster of) knowledge can have intrinsic methods. DOOD is a promising concept for the mechanism.

2.2 Requirements for DBMS

Among the requirements for databases we can find ones for DBMS or data models which require improvement in retrieval and identification. Traditional ways are not so appropriate for some molecular biological applications, e.g., sequence retrieval and quality management.

Information Retrieval

DBMS is expected to support the facilities of information retrieval for the sequences of DNA, RNA, and amino acids. It is partly because a concept of DBMS is rather wider for biologists than traditional one for database researchers.

[1]IntelliGenetics, Inc., Mountain View, CA
[2]National Institute of Health, Bethesda, MD
[3]Genetic Data Environment

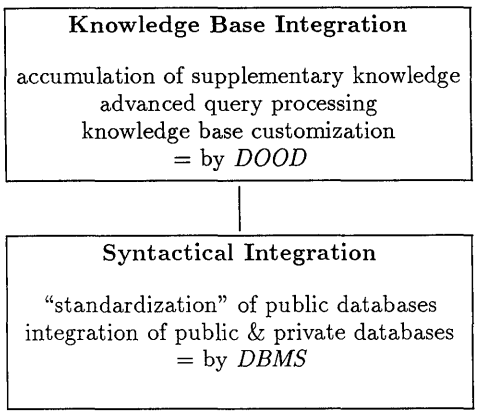


Figure 1: Integrated Knowledge Base of Proteins

Although sequence retrieval is like full text search, there are some differences in the search criteria: similarity search via dynamic programming (DP) or other algorithms (ex. BLAST[Altshul *et al.* 90]), with given similarities between characters (namely, amino acids).

"Keyword" extraction from the sequence is far more difficult than from the text. In order to process large sequences, they should be preprocessed by an information retrieval technique, as strings are preprocessed to make an alphabetical index. Keyword extraction corresponds to such preprocessing.

However, we should consider first, what word is in DNA or amino acid sequences. A gene is a sentence, and a DNA is a character. Thus, a word might be a specific DNA pattern closely related to some function, which is not so clear at present. An amino acid sequence is a sentence, and an amino acid is a character. So, a word may be a structural block or a functional block, either of which is represented as an amino acids' pattern including some varieties. Determining and extracting "keywords" is one of the big problems in biology.

At present, the sequences should be regarded as character strings, and not as paragraphs or sentences which consists of words. Moreover, we have to consider the following features of the sequences.

- DNA and RNA sequences consists of only four characters.
- Proteins mostly consist of 20 characters, but there are some exceptions.
- Similarity between the characters are defined.

Identification Facilities

DBMS is required to provide rich identification facilities. In treating molecular biological data, we should consider at least two kinds of errors: *experimental errors* and *identification errors*. Experimental errors are inevitable in molecular biological databases. It is necessary to repeat experiments to reduce them. In reading DNA sequences, for example, the *same region* should be repeatedly read to verify the result. In such case, GenBank is useful in reducing efforts of verifying. though it has sequences with various qualities. It contains many staff-reviewed sequences with many references, while it also contains a lot of sequences each of which has been just registered by a researcher.

Identification errors possibly occur in this verification process. It is not so clear whether we can get the sequence of the *same region* because of slightly different repeating regions, or natural errors such as diseases or mutations.

Representation of relations between proteins and functions are more ambiguous than relations between loci and DNA sequences in the example above. We should always consider identification errors in both proteins and functions. As experimental facts are accumulated, for example, "cytochromes transfer electrons" may turn into relations such as "cytochromes and ubiquinone transfer electrons" (protein identification is relaxed) or "cytochromes transfer electrons to generate energy" (function identification is detailed).

A concept of object identity is important in such cases. Results including experimental errors should be treated as different objects to store them avoiding integrity problems, whereas they should be treated as an object when we ask the "verified" result. Furthermore, identifiers should be so flexible that we can change them with as few difficulties as possible.

3 Protein Databases

In Section 2, general issues on molecular biological databases and DBMS are shown. This section focuses on protein databases and overviews the reasons for using existing public databases, as well as their general use, in order to consider the necessity of an integrated knowledge base.

3.1 Public Protein Databases

Protein information includes amino acid sequences, 3D structures, and functions. Protein functions include thermodynamic, chemical, and organic functions of total or partial proteins. In addition, there is important auxiliary information such as the authors, titles, and journals of the references relating to the data, source organisms, and experimental conditions.

Public protein databases have been trying to cover all protein information: amino acid sequences (PIR, Swiss-Prot), structures (PDB), partial patterns (ProSite), enzyme functions, and restricted enzymes (REBASE). All databases contain the auxiliary information mentioned

above.

The amount of information contained for each area is shown in Table 1. It seems possible that whole databases can be held privately in order to catalyze a change in their use: from databases accepting biological applications to knowledge bases including public databases and processing advanced biological queries.

Table 1: Public Protein Databases

database	release	entries	size
PIR	30.0 (9/91)	33,989	9,697,617 residues
Swiss-Prot	18.0 (5/91)	20,772	6,792,034 residues
PDB	(7/91)	688	153 Mbytes
ProSite	7.0 (5/91)	508	1 Mbytes
REBASE	(1/92)	1975	16 Mbytes

3.2 Purposes of Protein Databases

The final goal of using protein databases is to predict the unknown functions of a protein. Biologists gather enough of a known relations between functions and proteins to predict the unknown functions of a known/unknown protein as accurately as possible.

Its subgoals are important for molecular biology:

- Understanding of the relation between protein structure and function

  Most of protein functions are due to their structure. Structure might be predicted from their amino acid sequences, by *molecular dynamics* or several kinds of empirical approaches.

- Prediction of the 3D structure from the amino acid sequence

  Theoretically, most of protein 3D structures are calculated by molecular dynamics. It costs, however, enormous time to compute at present. Thus various empirical approaches have been tried, and would be tried.

The sequence similarity search, especially for extraction of common sequence patterns or similar regions (which are often called 'consensus sequences' in molecular biology) is a first step of empirical approaches. How to represent and how to use biological knowledge to predict unknown structures or unknown functions are other early problems.

3.3 General Use of Protein Databases

Similarity Search

Most traditional uses of protein databases are supported by traditional DBMS, except for similarity searches in the sequence database. Biologists ask the database to get a set of proteins whose name is, for example, "cytochrome c", or proteins which are found in "E.Coli." This type of retrieval is supported by the traditional DBMS.

They often want to examine such a set of sequences to discover a description of the similarity of a certain protein set, such as the existence of consensus sequences. They use *multiple alignment* [Ishikawa *et al.* 92] after they get all sequences they want. In this case, we consider interaction between application and database in rather higher level (see Section 6).

Another important use is similarity search. They search for amino acid sequences in the database that are similar to the unknown sequence they have. The unknown sequence may be a fragment or a whole sequence. The former is *motif search*, which is regarded as text content search, while the latter is *homology search*.

Biologists want to get accurate results in these searches and examination. Because the accuracy affects the quality of function prediction and structure prediction. They would like to retrieve the several of the best sequences of similar functions in the database.

In order to improve recall and precision ratios in protein similarity search, plenty of biologists' empirical knowledge and experimental results are indispensable. In addition to them, two problems have to be solved: finding an efficient algorithm for the homology and motif searches, and speeding up basic retrieval. The former needs the cooperation of biologists and computer scientists, whereas the latter could be devised independently by computer scientists, for some basic operations might be taken from techniques of the partial string match in the text database.

Data management

Data management, such as designing schema, storing data, and checking integrity, are owing to great efforts of the staffs of public databases.

Recently, schema of existing public databases are gradually standardized (as shown in Section 2), however, each existing database still employs independent naming rules using alphanumeric symbols such as 'P08478'(PIR), 'AMD1$XENLA'(Swiss-Prot), and '1.14.17.3'(Enzyme DB). Biologists are annoyed by updating cross-references among public databases and private ones.

As for storing, public databases accept an electronic form of registration to reduce staffs' efforts for quick storing. The U.S. National Institute of Health proposes a standard format for data exchanging (ASN.1 [NCBI 90]), which simplifies registration procedures and

is useful in gathering them into personal systems.

In order to distribute recent data as quickly as possible, PIR distributes less verified data for biologists. Thus, it reduces staffs' efforts for quick checking. When such data are used, verification process is owing to the biologists who would like to use them. PIR has three kinds of indications by their verification level: 'Annotated and Classified', 'Preliminary', and 'Unverified'. It is obvious that such indications are not enough for biologists' private data management.

Cooperation with biologists is indispensable in settling how to identify data with their quality and make cross reference data, although some management can be independently devised for advanced uses.

4 Kappa-P: An Extensible Parallel DBMS

We use Kappa-P as an ingredient in our integrated system. Kappa-P provides several facilities suitable for protein information. The efficiency of the nested relational model of Kappa is shown in [Yokota *et al.* 89], where efficient usage of storage and flexibility of schema evolution are described. In this section, we show the effectiveness of Kappa-P as an extensible DBMS for protein information processing and how to embed information retrieval facilities into Kappa-P.

4.1 Parallel DBMS

As the sequence search is executed exhaustively on a full sequence, its parallel processing is obviously effective. Fig. 2 shows the cofiguration of our system.

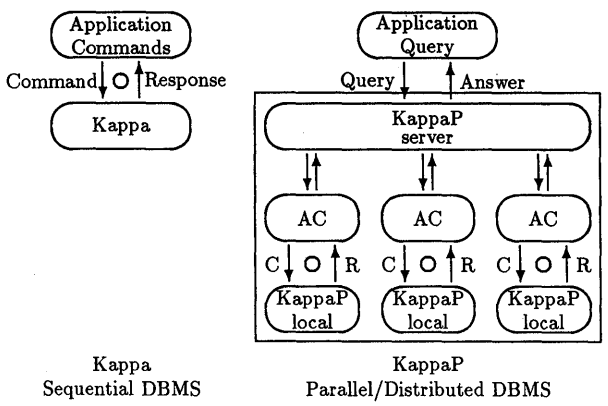


Figure 2: Configuration of Kappa-P

The aim of the extensibility of Kappa (sequential DBMS) is to reduce interaction with applications, and to customize the command interface for each application. The modules of an application which frequently use Kappa commands are included in Kappa system so that the number of communication among processes on the left-hand side of Fig. 2 decreases.

Beside these facilities, another kind of extensibility must be considered in a parallel system. Parallel DBMS Kappa-P consists of server DBMS and local DBMS, where server DBMS has a global map of local DBMSs and coordinates local DBMSs to deal with users' request, while local DBMS holds users' data [Kawamura *et al.* 92]. In our environment, most applications work on the same PIM with Kappa-P. So, if the server DBMS merges all the answers from local DBMSs into one answer, the effectiveness of parallel processing is reduced.

In order to avoid such a situation, the user defined commands, for example, DP or BLAST, are thrown to every local DBMS, and they play a role of filter from local databases to their server. The filters select data satisfying the given conditions, and send them to the server processor.

It is obviously efficient to throw application procedures to every local DBMS. The extensibility in Kappa contributes to efficient parallel processing of sequence search as in Fig. 2.

4.2 Information Retrieval

Extensible DBMS is also suitable for supporting information retrieval, obviously because it allows to customize command interface for applications. Sequence similarity searches, which correspond to full text searches, are implemented easily as "Application Commands (AC)" in Fig. 2.

We have developed a character-pair based index system, especially for motif search. This kind of index system is also implemented as application commands, while indexes are held in each local DBMS. Thus, the number of communications between local and server DBMS decreases.

Motif dictionary such as the public database ProSite could also be used as another useful index for sequence similarity searches. Extensible DBMS is so flexible that when such an improved index is developed it could be easily added in the system.

5 *QUIXOTE*: A Deductive Object Oriented Database

We use *QUIXOTE* as another ingredient in our system. Though advanced query processing is available by any logic programming language, facilities of *QUIXOTE* are more suitable to represent protein information, especially protein functions [Tanaka 91].

In this section we focus on its representation facilities in data management: schema flexibility and powerful identification.

5.1 Objects and Modules of quixote

Object Identifier

Objects in $\mathcal{QUIXOTE}$ are represented by extended terms called *object terms* [Yasukawa *et al.* 92]. An object term is of the following form:

$$o[l_1 = v_1, \ldots, l_n = v_n]$$

where o is called the *head* of the object term, l_i is a label, and v_i is the value of the l_i of the object term.

The labels and their values of an object term represent the *properties* of the object which are intrinsic to identify it. In such sense, object terms play a role of *object identifiers*. An object may have properties other than those specified in its object term. To represent such properties (extrinsic properties) of an object, special form of term representation called an *attribute term* is used:

$$o[l_1 = v_1, \ldots, l_n = v_n]/[l'_1 = v'_1, \ldots, l'_m = v'_m].$$

This attribute term represents that the object identified by the object term $o[l_1 = v_1, \ldots, l_n = v_n]$ has properties $[l'_1 = v'_1, \ldots, l'_m = v'_m]$. It is important to distinguish intrinsic properties with extrinsic ones.

Simplified examples are shown in Fig. 3. (1) and (2) describe the same object and their attribute terms contradict each other, while (1') and (2') represent different objects.

```
object_head [ol1 = ov1, ol2 = ov2, ...] /
            [al1 = av1, al2 = av2, ...]

  (1)  fact [label1=v1] / [label2=v2].
  (2)  fact [label1=v1] / [label2=v3].

  (1') fact [label1=v1, label2=v2].
  (2') fact [label1=v1, label2=v3].
```

Figure 3: Examples of $\mathcal{QUIXOTE}$ objects

Such representation of protein information is quite useful, for only the attributes whose values are determined can be used for identification. It is also useful in repeating local integrity checking, as data set would not stop increasing in amount.

Module

Modules in $\mathcal{QUIXOTE}$ help object management. Simplified examples are shown in Fig. 4.

```
(1)  module1 :: object1.

(2)  module2 >- module1.
(3)  module2 :: {{ object2. object3. }}

(4)  module3 >- module1.
(5)  module3 :: object3.
```

Figure 4: Examples of $\mathcal{QUIXOTE}$ modules

"`object1` is an object in `module1`" is represented as (1). (3) is an abbreviation form. (2) represents an order between modules specifying that `module2` inherits all the objects from `module1`, and (4) represents another inheritance. Therefore, `module2` has `object1`, `object2`, and `object3`, whereas `module3` has `object1` and `object3`.

Although `object3` is in both `module2` and `module3`, it may have different properties in each module, because any relations between `module2` and `module3` is not defined. We can give different properties to the same object in different modules. Thus, we can use different modules to avoid database inconsistency when we get different results by different experiments.

5.2 Identifiers of Proteins

Requirements

Since it is impossible to give the clearest identifier instantly, identification requires that the following be satisfied.

(1) Subsumption relation

An identifier sometimes has to be generalized or specialized. For example, the sentence "cytochromes have a certain feature" sometimes has to be reconsidered as "cytochromes and hemoglobins have a certain feature" or "cytochrome c has a certain feature." It seems rare to misidentify completely different objects. Most erroneous identifiers have to change only their abstraction level, and need not to be altered completely.

(2) Flexibility

In the process of determining the clearest identifier, we feel it useful if DBMS accepts tentative identifiers which can be specialized or generalized at anytime. We could use trial and error to determine the proper identifier.

(3) Module

To distinguish tentative identifiers from fixed ones, or experimental results from derived ones, a facility for making modules is required. It allows local integrity to be checked within the module, and for the

global uniqueness of the labels of the identifiers to be ignored.

Flexibility along Subsumption Relation

Proteins need identification transition along subsumption relation, as shown above. Fig. 5 is an example of how they are represented in *QUIXOTE*.

```
(fact1) cytochrome[lifename= E.Coli] /
                        [feature = featX].
(fact2) cytochrome[type= c] /
                        [feature = featX].
(hyp1)  cytochrome / [feature = featX].

(cf.1)  cytochrome( E.Coli, _, featX )
        cytochrome(      _, c, featX )
(cf.2)  cytochrome( [lifename(E.Coli),type(c)],
                    featX )

(hyp2) protein[name={cytochrome,hemoglobin}] /
              [feature = featX].
```

Figure 5: Proteins as objects in *QUIXOTE*

Provided that there is a feature named 'featX'. As experiments are repeated, the identifier of the protein whose feature is 'featX' may be changed.

QUIXOTE expressions (fact1) and (fact2) are examples of the identification of experimental results. (fact1) mentions nothing about the (fact2) attribute "type", and vice versa.

In the relational data model or Prolog, it is necessary to redesign the attributes of tables or arguments of facts to reflect such schema changes, since attributes have to be fixed. This is shown in (cf.1). In Prolog, we can reflect them by using lists as shown in (cf.2) [Yoshida *et al.* 91]. However, it is necessary to support a particular unifier for the list, and users must manage the meaning of the list (e.g., connected by 'and' or 'or') carefully. *QUIXOTE* allows set concepts with particular semantics to avoid such mismatches.

When we consider what sort of protein has 'featX' and get (fact1) and (fact2), we can easily think of a hypothesis (hyp1). We can also get this hypothesis by *QUIXOTE* , using object lattice of subsumption relation.

Moreover, if we give some relations among 'cytochrome', 'hemoglobin', and 'protein', another hypothesis such as (hyp2) is available.

Modules for Data Management

Objects of experimental results, verified results, and public databases have to be distinguished by modules, to be checked by different integrity checking methods.

Fig. 6 shows an example of verification process. Upper modules inherit all facts and rules of their lower modules. '**PIR**', '**Swiss-Prot**', and '**Experimental Results**' are modules each of which allows local integrity checking. If identifiers are conflicted between these modules, they can be settled at their upper module.

'**Sequence**' has some rules and cross-references between PIR and Swiss-Prot so that it can select and reply a specific set of protein sequences contained in these public databases. '**Integrated**' has some rules to verify experimental results by merging the selected sequences from public databases. It also has cross-references between public databases and experimental results (but they are ignored in Fig. 6 to simplify the example).

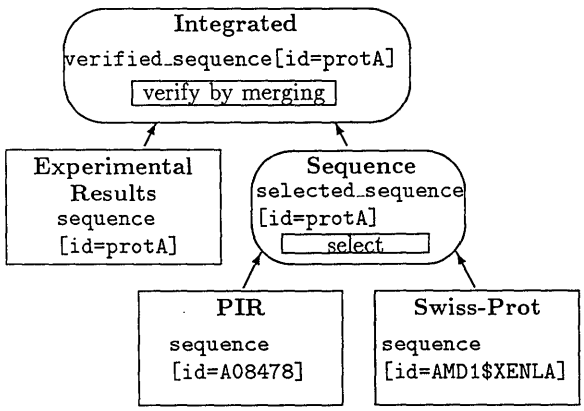


Figure 6: Modules for Verification Process

6 An Integrated System for an Integrated Knowledge Base

This section shows a system integration and a DB-KB integration, as to their configuration and their uses.

We are considering two kinds of integration: Kappa-P and *QUIXOTE* (DBMS and KRL), and existing public databases and biological knowledge (DB and KB).

6.1 Configurations of Integration

DBMS and KRL

There are three interactions in the integrated system of a database management system Kappa-P and a knowledge representation language *QUIXOTE* (see Fig. 7).

(1) Interactions between Kappa-P and *QUIXOTE*

All facts (non-temporal objects) in *QUIXOTE* are stored in Kappa-P, and Kappa-P activates necessary objects as the result of retrieval.

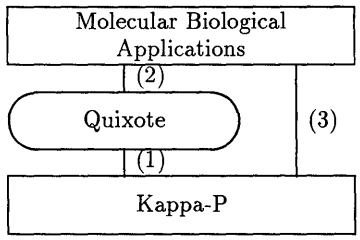


Figure 7: Integrated System of Kappa-P and Quixote

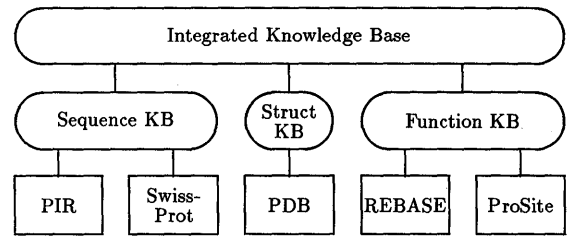


Figure 8: Integrated Knowledge Base of Proteins

(2) Applications and *QUIXOTE*

The followings are supported or will be provided (are under development).

- advanced query processing facility (inference)
  As *QUIXOTE* is an extension of Prolog, it provides more flexible and powerful query processing facility.
- a standard of the molecular biological data
  New databases and new rules (knowledge) are easily available by supporting ASN.1.
- graphical user interfaces
  Ad-hoc uses are quite important for biologists. The system should support ad-hoc queries, with graphical user-friendly interfaces. Kappa supports user interfaces for the nested relation and for PIR on X-window.
- class libraries for biological use
  This would include sequence retrieval and data management (see Sections 4 and 5).

(3) Applications and Kappa-P

The system should support direct access to databases for simple queries. It currently supports a graphical user interface to access amino acid sequences and some libraries to maintain biological data.

Protein Databases and Knowledge Bases

There are many public protein databases (see Section 3). We are holding several databases including such public ones as those shown at the bottom of Fig. 8.

An oval represents a module of rules and facts, while a rectangle represents a Kappa-P database. Modules in the upper two levels are mostly rules in *QUIXOTE*, while ones at the bottom are mostly facts in Kappa-P. The user may ask the top-level module any queries.

It can also be integrated with private databases and customized to be a private knowledge base. An example of such integration and customization is shown in Fig. 6.

6.2 Use of the System

Application of Sequence Analysis

Ishikawa et al. (ICOT) have developed a parallel processing algorithm of protein multiple alignment [Ishikawa *et al.* 92]. When the multiple alignment system and the knowledge base are connected, and a new multiple alignment algorithm using motifs is developed, it becomes an integrated application and knowledge base system. This is expected to enable automatic motif extraction and motif accumulation.

Advanced Query Processing

The query processing facilities of *QUIXOTE* realize a data pool of experimental results with query processing. They act as a prototype database or knowledge base for the experiment, which accumulates queries and shows the tendency of its usage in the integrated environment.

Graphical User Interface

The system has an user interface which allows it to use both an advanced query processing interface to *QUIXOTE* and a browsing and query-by-example interface for Kappa-P. The query interface provides or will provide facilities of displaying examples of queries, or graphs of answers such as the relations of objects given by a recursive query.

The browsing interface also provides or will provide graphical displaying facilities. We have developed a visual feature exhibition of sequences of both GenBank and PIR.

7 Conclusion

The requirements of molecular biology, especially protein engineering, which is a brand-new DBMS/KRL field were overviewed. Biological applications are now shown to be stimulating for DBMS and KRL, which are required to have various functions: information retrieval, deduction,

identification, module concepts, extensibility, and parallel processing.

Such facilities of DBMS/KRL had better be requested by (computer-)biologists. It is important to cooperate with them to conduct further research.

A private knowledge base including various existing public databases will proceed biological knowledge discovery. Although we have not mentioned in this paper, distributed DBMS is also necessary in case databases and knowledge bases exceed the personal system capacity. We think DOOD with extensible DBMS also play an important role, but it will be considered in future.

Acknowledgments

The author wishes to thank Kazumasa Yokota, Hideki Yasukawa, Moto Kawamura and other people in the QUIXOTE project and Kappa-P project for their valuable comments on earlier versions of this paper. The author also wishes to thank the people on the computer-biology mailing list for their suggestions from the viewpoint of biology.

References

[Altshul *et al.* 90] Altshul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J.: "Basic Local Alignment Search Tool", *J. Mol. Biol.* 215, pp.403-410, (1990).

[Ishikawa *et al.* 92] Ishikawa, M., Hoshida, M., Hirosawa, M., Toya, T., Onizuka, K. and Nitta, K.: "Protein Sequence Analysis by Parallel Inference Machine" *FGCS 92*, (Jun 1992).

[JIPID 90] PIR-International (JIPID): *PIR Newsletter*, No.3 (June 1990).

[Kawamura *et al.* 92] Kawamura, M., Sato, H., Naganuma, K. and Yokota, K.: "Parallel Database Management System : Kappa-P" *FGCS 92*, (Jun 1992).

[NCBI 90] NCBI: "GenInfo Backbone Database", Version 1.59, *Draft* (Apr 1990).

[Tanaka 91] Tanaka, H.: "Protein Function Database as a Deductive and Object-Oriented Database", *Database and Expert Systems Applications*, Springer-Verlag, pp.481-486 (Aug 1991).

[Yasukawa *et al.* 92] Yasukawa, H., Tsuda H. and Yokota, K.: "Objects, Properties, and Modules in Quixote", *FGCS 92*, (Jun 1992).

[Yokota *et al.* 89] Yokota, K. and Tanaka, H.: "GenBank in Nested Relation", *Joint Japanese-American Workshop on Future Trends in Logic Programming* (Oct 1989).

[Yoshida *et al.* 91] Yoshida, K., Overbeek, R., Zawada, D., Cantor, C.R. and Smith, C.L.: "Prototyping a Mapping Database of Chromosome 21", *Proceedings of Genome Mapping & Sequencing Meeting*, Cold Spring Harbor Laboratory, (1991).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers

Satoshi Terasaki, David J. Hawley*, Hiroyuki Sawada, Ken Satoh,
Satoshi Menju, Taro Kawagishi, Noboru Iwayama and Akira Aiba

Institute for New Generation Computer Technology
4–28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Abstract

Parallelization of a constraint logic programming (CLP) language can be considered at two major levels; the execution of an inference engine and a solver in parallel, and the execution of a solver in parallel. GDCC is a parallel CLP language that satisfies this two level parallelism. It is implemented in KL1 and is currently running on the Multi-PSI, a loosely coupled distributed memory parallel machine. GDCC has multiple solvers and a *block* mechanism that enables meta-operation to a constraint set. Currently there are three solvers: an algebraic solver for nonlinear algebraic equations using the Buchberger algorithm, a boolean solver for boolean equations using the Boolean Buchberger algorithm, and a linear integer solver for mixed integer programming. The Buchberger algorithm is a basic technology for symbolic algebra, and several attempts at its parallelization have appeared in the recent literature, with some good results for shared memory machines. The algorithm we present is designed for the distributed memory machine, but nevertheless shows consistently good performance and speedups for a number of standard benchmarks from the literature.

1 Introduction

Constraint logic programming (CLP) is an extension of logic programming that introduces a facility to write and solve constraints in a certain domain, where constraints are relations among objects. The CLP paradigm was proposed by Colmeraure[Colmerauer 87], and Jaffar and Lassez[Jaffar and Lassez 87]. A similar paradigm (or languages) was proposed by the ECRC group [Dincbas *et al.* 88]. A sequential CLP language CAL (*Contrainte avec Logique*) was also developed at ICOT[Aiba *et al.* 88].

The CLP paradigm is a powerful programming methodology that allows users to specify what (declarative knowledge) without specifying how (procedural knowledge). This abstraction allows programs to be more concise and more expressive. Unfortunately, the generality of constraint programs brings with it a higher computational cost. Parallelization is an effective way of making CLP systems efficient. There are two major levels of parallelizing CLP systems. One is the execution of an inference engine and constraint solvers in parallel. The other is the execution of a constraint solver in parallel.

Several works have been published on extending this work from the sequential to the concurrent frame. Among them are a proposal of ALPS[Maher 87] that introduces constraints into committed-choice language, a report on some preliminary experiments in integrating constraints into the PEPSys parallel logic system[Hentenryck 89], and a framework for a concurrent constraint (cc) language to integrate constraint programming with concurrent logic programming languages[Saraswat 89].

GDCC[Hawley 91b], Guarded Definite Clauses with Constraints, that satisfies two level parallelism, is a parallel CLP language that introduces the framework of cc into a committed-choice language KL1[Ueda and Chikayama 90], and is currently running on the Multi-PSI, a loosely coupled distributed memory parallel logic machine. GDCC has multiple solvers to enable a user to easily specify a proper solver for a domain: they are an algebraic solver, a boolean solver and a linear integer solver. The incremental evaluation facility is very important to CLP language solvers. That is, a solver must consider cases where constraints are dynamically added to it during execution, not only those cases where all are given statically prior to execution.

The algebraic solver is used to solve non-linear algebraic equations, and can be applied to fields such as computational geometries and handling robot design problems[S. Sato and Aiba 90]. The solver uses the Buchberger algorithm [Buchberger 83, Buchberger 85] that is a method of solving multi-variate polynomial equations. This algorithm is widely used in computer algebra, and also fits reasonably well into the CLP scheme since it is incremental and (almost) satisfaction-complete as shown in [Aiba *et al.* 88, Sakai and Aiba 89]. Re-

*Current office:: Compuflex Japan Inc. 12–4, Kasuya 4-chome, Setagaya-ku, Tokyo 157 Japan

cently, there have been several attempts made to parallelize the Buchberger algorithm, with generally disappointing results[Ponder 90, Senechaud 90], except for shared-memory machines[Vidal 90, Clarke *et al.* 90]. An interesting parallel logic programming approach implemented in Strand88[1] on Transputers was reported by Siegl[Siegl 90], with good speedups on the small examples shown, but absolute performance was only fair. We parallelize the Buchberger algorithm, emphasizing on absolute performance and incrementability rather than deceptive parallel speedups.

The boolean solver is used to solve boolean equations and can be applied to a wide range of applications such as logic circuit design. It uses the Boolean Buchberger algorithm [Y. Sato and Sakai 88]. It is different from the original Buchberger algorithm in load-balance of the internal processes, although they are basically similar. We implemented the parallel version of this algorithm, based on behavior analyses, using some example problems.

The target problems for the linear integer solver are combinatorial optimization problems such as scheduling problems, that obtain the minimum (or maximum) value with respect to an objective function in a discrete value domain under a certain constraint set. There are many kinds of formalization to solve the optimization problem, among them an integer programming that can be widely used for various problems. Integer programming still offers many methods of increasing search speed depending on the structures of problems, even if we focus on solving strictly optimized solutions only. The Branch-and-Bound method can apply to wide extent of problems independently to problem structures. We developed a parallel Branch-and-Bound algorithm, aiming to implement a high-speed constraint solver for large problems, and to perform experiments for describing parallel search problem in KL1.

The rest of this paper is organized as follows. We first mention the GDCC language and its system, and describe its parallel constraint solvers. Then, program examples in GDCC are shown using simple problems.

2 Parallel CLP Language

We will present a brief summary of the basic concepts of cc[Saraswat 89]. The cc programming language paradigm models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a (consistent) global database of constraints called the *store*. Constraints occurring in program text are classified by whether they are querying or asserting information, into the *Ask* and *Tell* constraints as shown in Figure 1.

[1]Strand88 is similar to KL1, although somewhat less powerful in that it does not support full unification.

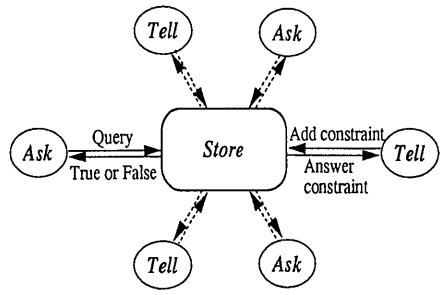


Figure 1: The cc language schema

This paradigm is embedded in a guarded (conditional) reduction system, where guards contain the *Ask* and *Tell*. Control is achieved by requiring that the *Ask* constraints in a guard are true (entailed), and that the *Tell* constraints are consistent (satisfiable), with respect to the current state of the *store*. Thus, this paradigm has a high affinity with KL1.

2.1 GDCC Language

GDCC is a member of the cc language family, although it does not support *Tell* in a guard part. The GDCC language includes most of KL1 as a subset; KL1 builtin predicates and unification can be regarded as the constraints of distinguished domain HERBRAND[Saraswat 89].

Now we define the logical semantics of GDCC as follows. S is a finite set of *sorts*, including the distinguished sort HERBRAND, F a set of *function symbols*, C a set of *constraint symbols*, P a set of *predicate symbols*, and V a set of *variables*. A sort is assigned to each variable and function symbol. A finite sequence of sorts, called a *signature*, is assigned to each function, predicate and constraint symbol. We define the following notations.

- We write $v : s$ if variable v has sort s,
- $f : s_1 s_2 \ldots s_n \rightarrow s$ if functor f has signature $s_1 s_2 \ldots s_n$ and sort s, and
- $p : s_1 s_2 \ldots s_n$ if predicate or constraint symbols p has signature $s_1 s_2 \ldots s_n$.

We require that terms be well-sorted, according to the standard inductive definitions. An *atomic constraint* is a well-sorted term of the form $c(t_1, t_2, \ldots, t_n)$ where c is a constraint symbol, and a *constraint* is a set of atomic constraints. Let Σ be the many-sorted vocabulary $F \cup C \cup P$. A *constraint system* is a tuple (Σ, Δ, V, C), where Δ is a class of Σ structures. We define the following meta-variables: c ranges over constraints and g,h range over atoms. We can now define the four relations *entails*, *accepts*, *rejects*, and *suspends*. Let x_g be the variables in constraints c and c_l.

Definition 2.1.1 *c entails* $c_l \stackrel{\text{def}}{=} \triangle \models (\forall x_g)(c \Rightarrow c_l)$

Definition 2.1.2 *c accepts* $c_l \stackrel{\text{def}}{=} \triangle \models (\exists)(c \wedge c_l)$

Definition 2.1.3 *c rejects* $c_l \stackrel{\text{def}}{=} \triangle \models (\forall x_g)(c \Rightarrow \neg c_l)$

Note that the property *entails* is strictly stronger than *accepts*, and that *accepts* and *rejects* are complementary.

Definition 2.1.4 *c suspends* c_l
$\stackrel{\text{def}}{=}$ *c accepts* $c_l \wedge \neg$ *(c entails* c_l *).*

A GDCC program is comprised of clauses that are defined as tuples (head, ask, tell, body), where "head" is a term having unique variables as arguments, "body" is a set of terms, "ask" is said to be *Ask constraint*, and "tell" is said to be *Tell constraint*. The "head" is the head part of the KL1 clause, "ask" corresponds to the guard part[2], and "tell" and "body" are the body part.

A clause (h, a, c, b) is a candidate for goal g in the presence of *store* s if $s \wedge g = h$ *entails* a. A goal g *commits* to candidate clause (h, a, c, b), by adding $t \cup c$ to the *store* s, and replacing g with b. A goal fails if the all candidate clauses are *rejected*. The determination of *entailment* for multiple clauses and *commitment* for multiple goals can be done in parallel.

Below is a program of pony_and_man written in GDCC.

```
pony_and_man(Heads,Legs,Ponies,Men) :- true |
    alg# Heads= Ponies + Men,
    alg# Legs=  4*Ponies + 2*Men.
```

Where, `pony_and_man(Heads,Legs,Ponies,Men)` is the head of the clause, "|" is the commit operator, `true` is an *Ask* constraint, equations that begin with `alg#` are *Tell* constraints. `alg#` indicates that the constraints are solved by the algebraic solver. In a body part, not only *Tell* constraints, but normal KL1 methods can also be written. In a guard part, we can only write read-only constraints that never change the content of the *store*, in the same way as the KL1 guard where active unification that binds a new value/structure to an undefined variable is inhibited.

But, bi-directionality in the evaluation of constraints, the important characteristic of CLP, is not spoiled by this limitation. For example, the query

```
?-  pony_and_man(5,14,Ponies,Men).
```

will return `Ponies=2, Men=3`. Thus, we can evaluate a constraint bi-directionally as *Tell* constraints have no limitations like *Ask*.

2.2 GDCC System

The GDCC system supports multiple plug-in constraint solvers with a standard stream-based interface, so that users can add new domains and solvers.

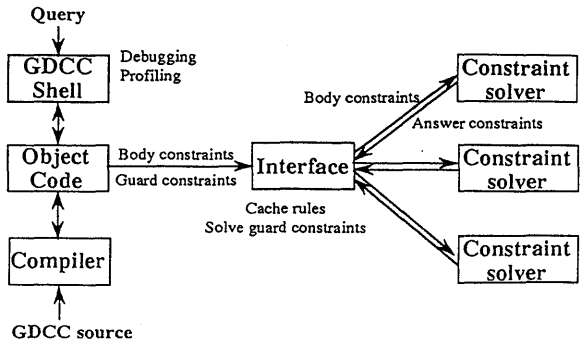


Figure 2: System Construction

The system is shown in Figure 2. The components are concurrent processes.

Specifically, a GDCC program and the constraint solvers may execute in parallel, "synchronizing" only and to the extent necessary, at the program's guard constraints.

The GDCC system consists of:

(i) Compiler
Translates a GDCC source program into KL1 code.

(ii) Shell
Translates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace and spy functions, together with solver-level event logging. The shell also provides limited support for incremental querying, in the form of inter-query variable and constraint persistence.

(iii) Interface
Interacts with a GDCC program (object code), sends body constraints to a solver and checks guard constraints using the results from a solver.

(iv) Constraint Solvers
Interact with the interface module and evaluate body constraints.

The decision of entailment using a constraint solver is described in each solver's section, as it differs from each algorithm adopted by a solver.

2.3 Block

A handling robot design support system [S. Sato and Aiba 90] has been used as an experimental application of our CLP systems for a few years. In applying GDCC to this problem, two problems arose. These were the handling of multiple contexts and the synchronization between an inference engine and solvers.

[2] "ask" contains constraints in the HERBRAND domain, that is, it includes the normal guards in KL1.

To clarify the backgrounds to these problems, we explain the handling of multiple contexts in sequential CLP language CAL. CAL has a function to compute approximated real roots in univariate non-linear equations. For instance, it can obtain values $X = \pm\sqrt{2}$ from $X^2 = 2$. Using this facility, the handling robot design support system can solve a given problem in detail. In this example, there are two constraint sets, one that includes $X = \sqrt{2}$, and another that includes $X = -\sqrt{2}$. CAL selects one constraint set from these two and solves it. Then the other is computed by backtracking (i.e., the system forces a failure). In other words, CAL handles these two contexts one-by-one, not simultaneously. In committed-choice language GDCC, however, we cannot use backtracking to handle multiple contexts. There are same problems in implementing hierarchical CLP language[K. Satoh and Aiba 90, K. Satoh 90b] in GDCC.

The other problem is the synchronization between an inference engine and solvers. It is necessary to describe to the timing and the target constraints to execute a function to find approximated real roots. In a sequential CLP, it is possible to control where this description is written in a program. While in GDCC, we need another kind of mechanism to specify a synchronization point, as a clause sequence in a program does not relate to the execution sequence. A similar situation occurs when a meta operation to constraint sets is required, such as computing a maximum value with respect to a given objective function.

Constraint sets in GDCC are basically treated as global. Introducing local constraint sets, however, independence of the global ones, can eliminate these problems. Multiple contexts are realized by considering each local constraint as one context. An inference engine and solvers can be synchronized at the end point of the evaluation of a local constraint set.

Therefore, we introduced a mechanism, called *block*, to describe the scope of a constraint set. We can solve a certain goal sequence with respect to a local constraint set. The block is represented in a program by a builtin predicate `call`, as follows.

`call(` *Goals* `)` `using` *Solver-Package* `for` *Domain*
`initial` *Input-Con* `giving` *Output-Con*

Constraints in goal sequence *Goals* are computed in a local constraint set. "`using` *Solver-Package* `for` *Domain*" denotes the use of *Solver-Package* for *Domain* in this block. "`initial` *Input-Con*" specifies the initial constraint set. "`giving` *Output-Con*" indicates that the result of computing in the block is *Output-Con*.

Both local variables and global variables can be used in a block where the local variables are only valid within the block and the global ones are valid even outside the block. Local variables are specified by the builtin predicate `alloc/2` that assigns variables to a block. Variables that are not allocated in a block are assumed to be global.

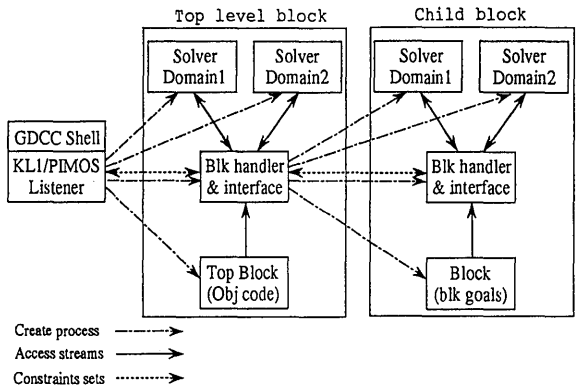


Figure 3: Implementation of *block* in GDCC

A block is executed by evaluating *Goals* with respect to *Input-Con*. The result of *Output-Con* is a local constraint set, that is, it is never merged with the global ones unless specified explicitly by a user.

Let us consider the next program.

```
test:- true |
    alloc(200,A),
    alg#A=-1,
    call( alg#A=1 ) initial nil giving C0,
    call( alg#A=0 ) initial nil giving C1.
```

This program returns the constraint set $\{A = 1\}$ as `C0` and the constraint set $\{A = 0\}$ as `C1`.

The block mechanism is implemented by the three modules shown in Figure 3; an inference engine(block), a block handler and constraint solvers. To encapsulate failure in a block, the *shoen* mechanism of PIMOS[Chikayama *et al.* 88] is used. The block handler creates a block process, sends constraints from a block to a constraint solver, and goals to other processors. Each GDCC goal has a stream connecting to the block handler to which the goal belongs.

3 Parallel Constraint Solvers

3.1 Algebraic Solver

3.1.1 Domain of Constraint

A constraint system that is the target domain of the algebraic solver is generally called a *nonlinear algebraic polynomial equation*. According to the definitions in Section 2.1, this can be formalized as the constraint system $(\Sigma = F \cup C \cup P, \triangle, V, C)$, where:

$$S = \{\mathbf{A}\}$$
$$F = \{\times : \mathbf{AA} \to \mathbf{A}, + : \mathbf{AA} \to \mathbf{A}\} \cup \{\text{fraction} :\to \mathbf{A}\}$$

C = {=}
P = {string starting with a lowercase letter}
V = {string starting with an uppercase letter}
$\triangle$ = axioms of complex numbers

with the structure

$D(\mathbf{A})$ = set of all algebraic numbers
$D(\times)$ = multiplication
$D(+)$ = addition
$D(\text{fraction})$ = rational number it denotes

3.1.2 Gröbner Basis and Buchberger Algorithm

Below is a brief introduction to some notation and definitions needed to explain Gröbner bases and the Buchberger algorithm. Then, the sequential version of the Buchberger algorithm, on which the parallel version is based, is presented.

Definition 3.1.1 (Power product, monomial) *Power product is a product comprised of nonzero and finite number of variables, that is,*

$$x_1 x_2 \ldots x_n \quad (n \geq 0, \text{ each } x_i \text{ are variable}).$$

Monomial is a product of a coefficient ($\in$rational number) and a power product.

A power product that contains no variable is written as "1".

Definition 3.1.2 (Admissible order) *An ordering $\prec$ is admissible when it satisfies the next properties. For all power products p, q, r,*

(i) $1 \prec p$, *and*

(ii) $p \prec q \Rightarrow pr \prec qr$.

Examples of admissible ordering that are often used in the Buchberger algorithm are *total degree lexicographic ordering* and *total degree reverse lexicographic ordering*. Let us represent the power product $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ by the vector $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$, where the variables are arranged in lexicographic order. We define the *total degree lexicographic order* $\prec_{dl}$ as follows.

$$\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \prec_{dl} \langle \beta_1, \beta_2, \ldots, \beta_n \rangle$$
$$\Leftrightarrow \sum_{i=1}^{n} \alpha_i < \sum_{i=1}^{n} \beta_i, \quad \text{or,}$$
$$\Leftrightarrow \sum_{i=1}^{n} \alpha_i = \sum_{i=1}^{n} \beta_i, \quad \exists i \ \ \alpha_i < \beta_i, \ \alpha_j = \beta_j \ (j < i).$$

That is, the order $\prec_{dl}$ determines a greater monomial by comparing the vector elements in lexicographic order, when the total degree is the same between the two monomials. On the other hand, the *total degree reverse lexicographic order* $\prec_{drl}$ is defined by:

$$\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \prec_{drl} \langle \beta_1, \beta_2, \ldots, \beta_n \rangle$$
$$\Leftrightarrow \sum_{i=1}^{n} \alpha_i < \sum_{i=1}^{n} \beta_i, \quad \text{or,}$$
$$\Leftrightarrow \langle \sum_{i=1}^{n} \alpha_i, -\alpha_n, \ldots, -\alpha_2 \rangle \prec_{dl} \langle \sum_{i=1}^{n} \beta_i, -\beta_2, \ldots, -\beta_2 \rangle$$

When the total degree of two monomials is equal, this order compares the subtotal degree by removing the last elements from both vectors.

Let $Lt(f)$ denote the maximal monomial of a polynomial f with respect to a certain admissible ordering, and $Rest(f)$ mean the remaining monomials of f. Let the power product and coefficient of $Lt(f)$ be $Lp(f)$ and $Lc(f)$ respectively.

For each polynomial f $(= Lc(f)Lp(f) + Rest(f))$, we define a rewriting rule $\Rightarrow_f$ over polynomials as follows.

Definition 3.1.3 (Rewriting) $g \Rightarrow_f h$, *if a monomial of a polynomial p is a multiple of $Lp(f)$ then the monomial is replaced with $\frac{-Rest(f)}{Lc(f)}$, and the result of calculation by the replacement is h. For a finite set of polynomials G, $g \Rightarrow_G h$ if $\exists f \in G$ and $g \Rightarrow_f h$.*

Definition 3.1.4 (Irreducible) *The irreducible form of a polynomial g w.r.t. $\Rightarrow_G$ is the polynomial which cannot be rewritten by $\Rightarrow_G$ any more after applying the rewriting rule set G finitely many (or zero) times. The irreducible form of g is denoted by $g\downarrow_G$.*

Let $R[x_1, \ldots, x_m]$ be a polynomial ring in n variable of $x_1, \ldots, x_m$ over the rational number field, and $f_1, \ldots, f_n$ be elements of it. A polynomial ideal $\mathcal{I}$ generated by $f_1, \ldots, f_n$ is a polynomial set defined by the following.

Definition 3.1.5 (Polynomial ideal)

(i) $\mathcal{I} \neq \phi, \quad f, g \in \mathcal{I} \Rightarrow f - g \in \mathcal{I}$ *(property of modules)*

(ii) $f \in \mathcal{I} \Rightarrow h \cdot f \in \mathcal{I}$ *for any* $h \in R[x_1, \ldots, x_m]$

With no loss of generality, we can assume that all polynomial equations are in the form $f = 0$. Let $E = 0$ be a system of polynomial equations $\{f_1 = 0, \ldots, f_n = 0\}$. The following close relation between the solutions of $E = 0$ and the elements of $\mathcal{I}(E)$ of the ideal generated by E is well known.

Theorem 3.1.1 (Hilbert zero point theorem) *Let f be a polynomial. Every solution of $E = 0$ is also a solution of $f = 0$, iff there exists a natural number s such that $f^s \in \mathcal{I}(E)$.*

Corollary 3.1.1 *E has no solution iff $1 \in \mathcal{I}(E)$.*

Thus, the problem of solving given polynomial equations is reduced to that of deciding whether a polynomial belongs to the ideal. Buchberger introduced the notion of Gröbner bases, and devised an algorithm to determine the membership relations of a polynomial and to the ideal [Buchberger 83, Buchberger 85].

Let there be an admissible ordering among monomials and let a system of polynomial equations $E = 0$ be given.

A rough sketch of the algorithm is as follows. In the system of E, each equation can be considered as being a rewriting rule as defined in Definition 3.1.3. When the left hand sides $Lp(f_1)$ and $Lp(f_2)$ of two rewrite rules f_1 and f_2 are not mutually prime, the least common multiple of their left hand sides can be rewritten in two different ways according to these two rules. The pair resulting from this rewriting is called a critical pair. If further rewriting does not succeed in converging a critical pair, the pair is said to be divergent. To get a confluent rewriting system, equations made from such critical pairs, S-polynomials, are added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting rule set. This confluent rewriting rule set is called a Gröbner basis of E.

Definition 3.1.6 (Gröbner basis [Buchberger 83]) *The Gröbner basis $G(E)$ is a finite set that satisfies the following properties.*

(i) $\mathcal{I}(E) = \mathcal{I}(G(E))$

(ii) For all f, g, $f - g \in \mathcal{I}(E)$ iff $f\downarrow_G = g\downarrow_G$, *especially,* $f \in \mathcal{I}(E)$ iff $f\downarrow_G = 0$, *and,*

(iii) G *is* reduced *if every element of the basis is irreducible w.r.t. all the others.*

From Theorem 3.1.1, the reduced $G(E)$ can be regarded as being the canonical form of the solution of $E = 0$, because the reduced Gröbner basis with respect to a given admissible ordering is unique. Moreover, when $E = 0$ does not have a solution, $\{1\} \in G(E)$ is deduced from Corollary 3.1.1.

Definition 3.1.7 (Critical pair, S-polynomial) *If two rewriting rules f_1, f_2 are not mutually prime, that is $Lp(f_1)$ and $Lp(f_2)$ have a greatest common divisor other than* 1, *the pair f_1, f_2 is called the* critical pair, *and the polynomial made from this critical pair in the following way:*

$$Lc(f_2)\frac{lcm(f_1,f_2)}{Lp(f_1)} \cdot f_1 - Lc(f_1)\frac{lcm(f_1,f_2)}{Lp(f_2)} \cdot f_2$$

is called S-polynomial *and denoted by $Spoly(f_1, f_2)$. where, $lcm(f_1, f_2)$ is the least common multiple of $Lp(f_1)$ and $Lp(f_2)$.*

Figure 4 shows the sequential version of the Buchberger algorithm. E denotes the input polynomial equation set, and R is the output Gröbner basis. Line (4) indicates the rewriting process using R. Lines (7), (8) and (9) are the subsumption test in which the old rule set is updated by the newly generated rule. If the left hand side of an old rule is rewritten by the new rule, the rewritten rule goes back to equation set F. Line (12) is the S-polynomial generation.

```
(1)  input F := E, R := ∅
(2)  while F ≠ ∅
(3)    choose f ∈ F
(4)    F := F − {f},  f' := f ↓R
(5)    if f' ≠ 0 then
(6)      for every p ∈ R
(7)        if Lt(p) ⇒f' lt(p')
(8)          then F := F ∪ {lt(p')+Rest(p)},  R := R − {p}
(9)          else R := (R − {p}) ∪ {Lt(p)+Rest(p)↓R∪{f'}}
(10)       endif
(11)     endfor
(12)     F := F ∪ Spoly(f', R)†,  R := R ∪ {f'}
(13)   endif
(14) endwhile
(15) output R (R is G(E))
```

†: $Spoly(f', R)$ is to be generated by S-polynomials between polynomial f' and all elements in rule set R.

Figure 4: Sequential Buchberger algorithm

3.1.3 Satisfiability, Entailment

Based on the above results, we could determine satisfiability by using the Buchberger algorithm to incorporate the polynomial into the Gröbner bases as per Corollary 3.1.1. But the method of Definition 3.1.6(ii) is incomplete in terms of deciding entailment, since the relation between the solutions and the ideal described in Theorem 3.1.1 is incomplete. For example, the Gröbner basis of $\{X^2 = 0\}$ is $\{X^2 \to 0\}$, and rewriting using this Gröbner basis cannot show that $X = 0$ is entailed. There are several approaches solving the entailment problem:

(a) Use the Gröbner basis of the radical of the generated ideal, $\mathcal{I}$, i.e. $\{p | p^n \in \mathcal{I}\}$. Although it is theoretically computable, efficient implementation is not possible.

(b) As a negation of $p = 0$, add $p\alpha$ to the Gröbner basis and use the Buchberger algorithm, where α is a new variable. Iff 1 is included in the new Gröbner basis, $p = 0$ is held in the old Gröbner basis. This has the unfortunate side-effect of changing the Gröbner basis.

(c) Find n such that p^n is rewritten to 0 by the Gröbner basis of the generated ideal. Since n is bounded[Cangilia *et al.* 88], this is a complete decision procedure. The bound, however, is very large.

When there are a lot of resources to compute, and no more computation can be done, according to the method described in (c) we may adopt the incremental solution of repeatedly raising p from a small positive integer power and rewriting it by the Gröbner basis. On the other hand, the total efficiency of the system is greatly affected by the

computation time in deciding entailment. Therefore, we determine the entailment by rewriting using a Gröbner basis from the view point of efficiency, even though this method is incomplete. This decision procedure runs on the interface module parallel with the solver execution, as shown in Figure 2. Whenever a new rule is generated, the solver sends the new rule to the interface module via a communication stream. The interface determines entailment while storing (intermediate) rules to a self database. The interface updates the database by itself whenever a new rule from the solver arrives. It can also handle constraints such as inequalities in the guard parts, if they can be solved by passive evaluation.

3.1.4 Parallel Algebraic Solver

There are two main sources of parallelism in the Buchberger algorithm, the parallel rewriting of a set of polynomials, and the parallel testing for subsumption of a new rule against the other rules. Since the latter is inexpensive, we should concentrate on parallelizing the coarse-grained reduction component for the distributed memory machine. However, since the convergence rate of the Buchberger algorithm is very sensitive to the order in which polynomials are converted into rules, an implementation must be careful to select "small" polynomials early.

Three different architectures have been implemented; namely, a pipeline, a distributed, and a master-slave architecture. The distributed architecture was already reported in [Hawley 91a, Hawley 91b], however, it has been greatly refined since then. The master-slave architecture also offers comparatively good performance. Thus, we touch on the distributed and master-slave architectures in the following sections.

Distributed architecture

The key idea underlying the distributed architecture is that of sorting a distributed set of polynomials. Each processor contains a complete set of rewriting rules and polynomials, and a load-distribution function ω that logically partitions the polynomials by specifying which processor "owns" which polynomials. The position in the output rule sequence of each polynomial is calculated by its owning processor, based on an associated key (the leading power product), identical in every processor, and which does not change during reduction. A polynomial is output once it becomes the smallest remaining. The S-polynomials and subsumptions are calculated independently by each processor, so that the processors' sets of polynomials stay synchronized. As a background task, each processor rewrites the polynomials it owns, starting with those lowest in the sorted order. Termination of the algorithm is detected independently by each engine, when the input equation stream is closed, and when there are no polynomials remaining to be rewritten.

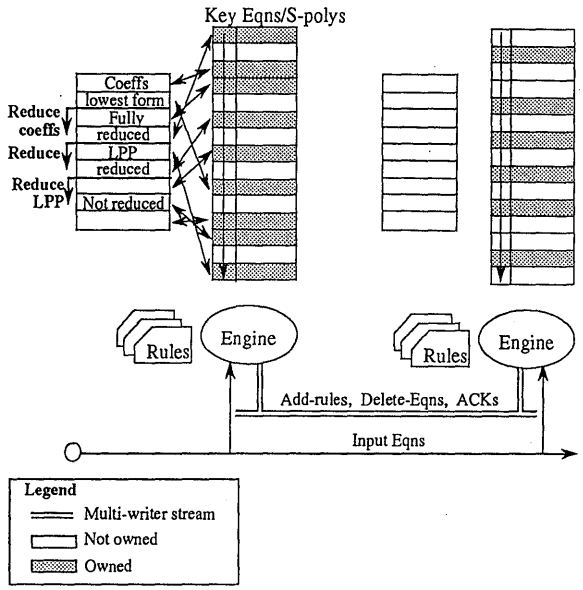


Figure 5: Architecture of distributed type solver

Figure 5 shows the architecture. The central data structures in the implementation are two work item lists: the global list and the local list. The global list, that contains all polynomials including both owned and not owned polynomials, is used to decide the order in which a processor can output a new rule based on the keys of polynomials. On the other hand, the local list consists of owned polynomials only. Items in the local list are rearranged by each processor to maintain increasing key order, whenever an owned polynomial is rewritten.

There will be a situation where, when a processor is busy rewriting polynomials, another processor outputs a new rule. In such a case, any processor that receives a new rule must quit the current task as soon as possible to check subsumption and to update the old rule set. Continuing tasks while using the old rule set without interruption increases the number of useless tasks. To manage such interruption and resumption of rewriting, the complete execution of one piece of work is broken down into a three-stage pipeline; first polynomials are rewritten until the leading power products can be reduced no further, they are fully reduced, and thirdly the coefficients are reduced by taking the greatest common divisor among all coefficients of a polynomial. Based on this breakdown, we pipeline the execution of the entire list, giving us maximum overlap between communication and local computation.

Table 1 shows the results of benchmark problems to show the performance of this parallel algorithm, the benchmark problems are adopted from [Boege *et al.* 86, Backelin and Fröberg91]. The monomial ordering is degree reverse lexicographic, and low level bignum (mul-

Table 1: Timing (sec) and speedup obtained with distributed architecture

Problems	Number of processors 1	2	4	8	16
Katsura-4	9.86	7.48	5.34	4.82	5.94
	1	1.32	1.85	2.05	1.66
Katsura-5	94.89	62.43	48.20	39.95	40.52
	1	1.52	1.97	2.38	2.34
Cyc5-roots	37.24	33.33	20.02	22.52	29.73
	1	1.12	1.86	1.65	1.25
Cyc6-roots	1268.96	1396.37	1555.58	817.07	3266.68
	1	0.909	0.816	1.55	0.388

tiple precision integer) support on PIMOS is used for coefficient calculation. The method of detecting unnecessary S-polynomials proposed by [Gebauer and Möller 88] is implemented. Examples and their variable ordering are shown below.

Katsura-4: $(U_0 < U_1 < U_2 < U_3 < U_4)$

$$U_0^2 - U_0 + 2U_1^2 + 2U_2^2 + 2U_3^2 + 2U_4^2 = 0$$
$$2U_0U_1 + 2U_1U_2 + 2U_2U_3 + 2U_3U_4 - U_1 = 0$$
$$2U_0U_2 + 2U_1^2 + 2U_1U_3 + 2U_2U_4 - U_2 = 0$$
$$2U_0U_3 + 2U_1U_2 + 2U_1U_4 - U_3 = 0$$
$$U_0 + 2U_1 + 2U_2 + 2U_3 + 2U_4 - 1 = 0$$

Katsura-5: $(U_0 < U_1 < U_2 < U_3 < U_4 < U_5)$

$$U_0^2 - U_0 + 2U_1^2 + 2U_2^2 + 2U_3^2 + 2U_4^2 + 2U_5^2 = 0$$
$$2U_0U_1 + 2U_1U_2 + 2U_2U_3 + 2U_3U_4 + 2U_4U_5 - U_1 = 0$$
$$2U_0U_2 + 2U_1^2 + 2U_1U_3 + 2U_2U_4 + 2U_3U_5 - U_2 = 0$$
$$2U_0U_3 + 2U_1U_2 + 2U_1U_4 + 2U_2U_5 - U_3 = 0$$
$$2U_0U_4 + 2U_1U_3 + 2U_1U_5 + U_2^2 - U_4 = 0$$
$$U_0 + 2U_1 + 2U_2 + 2U_3 + 2U_4 + 2U_5 - 1 = 0$$

Cyclic 5-roots: $(X_1 < X_2 < X_3 < X_4 < X_5)$

$$X_1 + X_2 + X_3 + X_4 + X_5 = 0$$
$$X_1X_2 + X_2X_3 + X_3X_4 + X_4X_5 + X_5X_1 = 0$$
$$X_1X_2X_3 + X_2X_3X_4 + X_3X_4X_5 + X_4X_5X_1 + X_5X_1X_2 = 0$$
$$X_1X_2X_3X_4 + X_2X_3X_4X_5 + X_3X_4X_5X_1 + X_4X_5X_1X_2 + X_5X_1X_2X_3 = 0$$
$$X_1X_2X_3X_4X_5 = 1$$

Cyclic 6-roots: $(X_1 < X_2 < X_3 < X_4 < X_5 < X_6)$

$$X_1 + X_2 + X_3 + X_4 + X_5 + X_6 = 0$$
$$X_1X_2 + X_2X_3 + X_3X_4 + X_4X_5 + X_5X_6 + X_6X_1 = 0$$
$$X_1X_2X_3 + X_2X_3X_4 + X_3X_4X_5 + X_4X_5X_6 + X_5X_6X_1 + X_6X_1X_2 = 0$$
$$X_1X_2X_3X_4 + X_2X_3X_4X_5 + X_3X_4X_5X_6 + X_4X_5X_6X_1 + X_5X_6X_1X_2 + X_6X_1X_2X_3 = 0$$
$$X_1X_2X_3X_4X_5 + X_2X_3X_4X_5X_6 + X_3X_4X_5X_6X_1 + X_4X_5X_6X_1X_2 + X_5X_6X_1X_2X_3 + X_6X_1X_2X_3X_4 = 0$$
$$X_1X_2X_3X_4X_5X_6 = 1$$

Sometimes parallel execution is slower than sequential execution. Moreover a serious drawback occurs in the case of "cyclic 6-roots". The reasons are; first, redundant tasks increase in parallel since updating a rule set, generating S-polynomials and detecting unnecessary S-polynomials are overlapped with every processor, second, the selection criteria of the next new rule is only a rough approximation as the keys of not owned polynomials are never updated during rewriting.

Master-slave architecture

In the distributed architecture, if the keys of other polynomials are updated according to their rewriting such that the global smallest polynomial can be found, then much communication between the processors is required. One simple way of avoiding such communication overhead is to have each processor output the local minimum polynomial and another processor decide the global minimum among them. Our third trial, therefore, is the master-slave architecture shown in Figure 6.

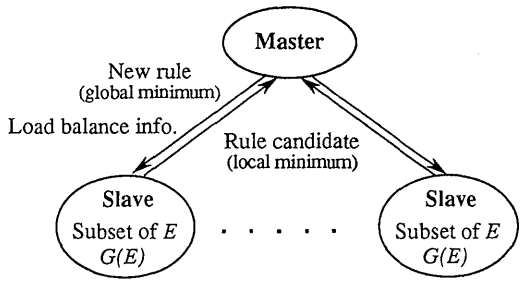


Figure 6: Architecture of master-slave type solver

The set of polynomials E is physically partitioned and each slave has a different part of them. The initial rule set of $G(E)$ is duplicated and assigned to all slaves. New input polynomials are distributed to the slaves by the master. The reduction cycle proceeds as follows.

Each slave rewrites its own polynomials by the $G(E)$, selects the local minimum polynomial from them, and sends its leading power product to the master. The master processor awaits reports from all the slaves, and selects the global minimum power product. The minimum polynomial can be decided only after all the slaves have reported to the master. Those that are not minimums can be decided quickly, however. Thus, the *not-minimum* message is sent to the slaves as soon as possible, and the processors receive the *not-minimum* message reduce polynomial by the old rule set while waiting for a new rule. On one hand, the slave that receives the *minimum* message converts the polynomial into a new rule and sends it to the master, the master sends the new rule to all the slaves except the owner. If several candidates are equal power products, all candidates are converted to rules by owner slaves and they go to final selection by the master.

To make load balance during rewriting, each slave reports the number of polynomials it owns, piggybacked

onto leading power product information. The master sorts these numbers into increasing order and decides the order in which to distribute S-polynomials. After applying the unnecessary S-polynomial criterion, each slave generates the S-polynomials it should own corresponding to the order decided by the master. Subsumption test and rule update are done independently by each slave.

Table 2 lists the results of the benchmark problems. The monomial ordering, bignum support and variable ordering are same as for the distributed architecture. Both absolute performance and speedup are improved compared with the distributed architecture. Speedup appears to become saturated at 4 or 8 processors except for "cyclic 6-roots". However, these problems are too small to obtain a good speedup because it takes about half a minute until all the processors become fully operational as the unnecessary S-polynomial criterion works well.

Table 2: Timing and speedup of the master-slave architecture

	Number of processors				
Problems	1	2	4	8	16
Katsura-4 (sec)	8.90	7.00	5.83	6.53	9.26
	1	1.27	1.53	1.36	0.96
Katsura-5 (sec)	86.74	57.81	39.88	31.89	36.00
	1	1.50	2.18	2.72	2.41
Cyc.5-roots (sec)	27.58	21.08	19.27	19.16	25.20
	1	1.31	1.43	1.44	1.10
Cyc.6-roots (sec)	1430.18	863.62	433.73	333.25	323.38
	1	1.66	3.30	4.29	4.42

3.2 Boolean Constraint Solver

An algorithm called the Boolean Buchberger algorithm [Y. Sato and Sakai 88] has been proposed for boolean constraints. Boolean constraints are handled differently from algebraic constraints in the following points.

(i) Multiplication and addition are logical-and and exclusive-or, respectively, in boolean constraints.

(ii) Coefficients are boolean values, that is, 1 and 0. So, a monomial is a product of variables.

(iii) The power of a variable is equal to the variable itself ($X^n = X$). So, a monomial is actually a product of distinct variables.

From the property (iii), the theorem of a boolean polynomial that corresponds to Theorem 3.1.1 is as follows.

Theorem 3.2.1 (Zero point theorem) *Let f be a boolean polynomial. Every solution of $E = 0$ is also a solution of $f = 0$, iff $f \in \mathcal{I}(E)$.*

Therefore, the relation between an ideal and solution and the relation between a solution and a Gröbner basis is complete in a boolean polynomial. Thus, *entailment* can be decided by rewriting a guard constraint by a Gröbner basis.

The Boolean Buchberger algorithm differs from the (algebraic) Buchberger algorithm in the following points. That is, we have to consider *self-critical pairs* as well as critical pairs, where a *self-critical pair polynomial* (SC-polynomial) of boolean polynomial f is defined as $Xf + f$ for every variable X of $Lp(f)$. As shown (ii) above, the coefficient calculation in the boolean solver is much cheaper than the algebraic solver, while *self-critical pairs* have to be considered. Thus, the load-balance of this algorithm is completely different from that of the algebraic solver.

3.2.1 Analysis of Sequential Algorithm and Parallel Architecture

The sequential Boolean Buchberger algorithm is shown in Figure 7. Here *EQlist* is a list of input boolean constraints and *GB* is a Boolean Gröbner basis. Numbers (1) to (6) indicate the step number of the algorithm.

From Figure 7 we can see that the following are possible for parallel execution;

(i) polynomial rewriting in step 6,

(ii) monomial rewriting (lower granularity of (i)),

(iii) subsumption test in step 4,

(iv) SC-polynomial generation in step 5, and

(v) S-polynomial generation in step 5.

Since there is a communication overhead in the distributed memory machine, we have to exploit the most coarse-grained parallelism. To design a parallel execution model, we measured the execution time in each step in Figure 7 using two kinds of example program. One is a logic circuit problem for a counter circuit that counts the number of 1's in a three-bit input and outputs the results as a binary code. The other is the n-queens problem where 4 queens have 80 equations with 16 variables, 5 queens have 165 equations with 25 variables, and 6 queens have 296 equations with 36 variables. The time ratio for each step is shown in Table 3.

Table 3: Time ratio of each step (%)

	Step number						
Problem	1	2	3	4	5	6	Total(sec)
4queens	25.8	3.2	8.4	17.3	25.4	19.0	1.8
5queens	6.4	3.4	22.3	3.7	14.5	50.4	53.5
6queens	1.0	1.5	15.0	2.0	2.6	77.7	2240.0
circuit	2.1	4.2	8.2	3.3	8.0	74.0	70.7

```
input EQlist, GB
EQlist := {p ∈ EQlist | p ↓GB≠ 0}

while EQlist ≠ ∅
(1) { q := min{Lp(p) | p ∈ EQlist}
      choose e ∈ {p ∈ EQlist | Lp(p) = q}
      EQlist := EQlist − {e}
(2)   r = e ↓GB, RWlist := ∅
      for every p ∈ GB
        if Lp(p) ⇒r p'
          then GB := GB − {p}
               RWlist := RWlist ∪ {p' + Rest(p)}
(3)       else GB := (GB − {p})
                     ∪ {Lp(p) + Rest(p) ↓GB∪{r}}
        endif
      endfor
      GB := GB ∪ {r}
      for every p ∈ EQlist
        if Lp(p) ⇒r p'
          then EQlist := EQlist − {p}
(4)            RWlist := RWlist ∪ {p' + Rest(p)}
        endif
      endfor
(5)   RWlist := RWlist ∪ SCpoly(r)† ∪ Spoly(r, GB)
      while RWlist ≠ ∅
        choose p ∈ RWlist
        RWlist := RWlist − {p}
        if p ≠ 0
          then if Lp(p) ⇒GB p'
(6)         then RWlist := RWlist ∪ {p' + Rest(p)}
            else EQlist := EQlist ∪ {p}
          endif
        endif
      endwhile
endwhile
output GB
```

† : $SCpoly(r)$ indicates the set of all self-crirical pair polynomials for r.

Figure 7: Booelan Buchberger algorithm

We can consider another parallel execution model by modifying the algorithm. Although Figure 7 shows all the reducible polynomials lumped together and rewritten in step 6, this reduction may be distributed to steps 3, 4 and 5. Moreover, reduction may be done in each step independently. Let steps 3', 4' and 5' denote the modified steps 3, 4 and 5. If execution times of steps 3', 4' and 5' are balanced after applying the modification to the algorithm, this model is also a good parallel execution model. However, as shown in Table 4, the times are not balanced. So, we can discard this possibility of parallelization.

From the above analysis, it becomes clear that step 6 is the largest part of the execution, the other parts being small. Therefore, we can determine the master-slave parallel execution model to make the best use of parallelism in step 6, as shown in Figure 8.

Table 4: Time ratio in modified algorithm (%)

Problem	Step number 3'	4'	5'
4queens	12.1	23.5	36.4
5queens	24.7	11.2	54.0
6queens	15.5	39.9	41.9
circuit	8.2	33.5	51.7

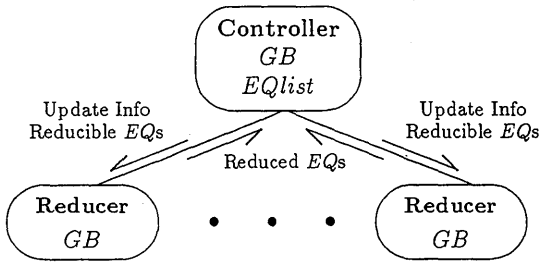


Figure 8: Parallel execution model

The controller (master) is in charge of step 1 to step 5 in the algorithm and the other reducers (slaves) reduce polynomials by GB. The message from the controller to the reducers consists of update information for GB and the polynomials to be rewritten. After receiving the message, the reducer first updates its current GB according to the update information, rewrites the polynomials from the controller, and finally sends the results of the reduction to the controller. As the controller becomes idle after sending the message, the controller also acts as a reducer during the reduction process. The number of polynomials sent to each reducer is kept as equal as possible to balance the loads for each processor.

3.2.2 Implementation and Evaluation

Having implemented the above parallel execution model in KL1, the following improvement was made.

Improvement 1 We can remove redundant equations from $EQlist$, produced by deleting rules in step 3, prior to their distribution. Although this removal can be done in each reducer, the distributed tasks may not be well balanced since the removal of tasks is much less involved than reduction.

Improvement 2 We can distinguish rules of the form "$x = A$" ("A" is variable) from other rules since these rules express assignments only and we need not consider SC-polynomials nor S-polynomials for these rules. These rules are stored differently in the controller and, if a new equation is input, we first apply these assignments in the controller to the equation.

By this application, reducers do not have to store such rules and the time needed to generate an SC-polynomial and S-polynomial can be saved.

Improvement 3 If the right hand side (RHS) of a rule is 0, then no SC-polynomial can be produced. If both RHSs of two rules are 0, then an S-polynomial cannot be produced. Therefore, the RHS of a rule is checked first. This technique is also effective for the sequential version.

Table 5 lists the execution times and the improvement ratio for the 6 queens problem.

Table 5: Timing and improvement ratio

Number of PEs	1	2	4	8	16
Original version (sec)	3735	2400	1745	1539	1262
Improved version (sec)	2489	1706	1223	1142	1092
Improvement ratio(%)	66.6	71.1	70.1	74.2	86.5

Let a purely sequential part in the parallel execution model be a and its parallel executable part be b. Then, we can approximate the execution time for n PEs as $(a+b)/n$. By calculating a and b from the data, we obtain $a = 1130$, $b = 2590$ for the original version, and $a = 930$, $b = 1540$ for the improved version. This means that the parallel executable part constitutes 70% to the entire execution for the original version and 62% for the improved version. Since we parallelized the sequential algorithm to obtain the original version, 70% is a satisfactory ratio for parallel execution since this ratio is very near to the upper bound value calculated from the analysis of the sequential algorithm. The difference is caused by the task distribution overhead. In the improved version, the ratio of the parallel executable part is decreased because of the increase in the number of controller tasks. However, this result is encouraging since the overall performance is improved.

3.3 Integer Linear Constraint Solver

The constraint solver for the integer linear domain checks the consistency of the given equalities and inequalities of rational coefficients, and gives the maximum or minimum values of the objective linear function under these constraint conditions. The integer linear solver utilizes the rational linear solver for the optimization procedure to obtain the evaluation of relaxed linear problems created as part of the solution. A rational linear solver is realized by the simplex algorithm. The purpose of this constraint solver is to provide a fast solver for the integer optimization domain by achieving a computation speedup by incorporating the search process into a parallel program.

These solvers can determine satisfiability and entailment. Satisfiability can be easily checked by the simplex algorithm. Entailment is equivalent to negation failure with respect to a constraint set.

In the following we discuss the parallel search method employed in this integer linear constraint solver. The problem we are addressing is a mixed integer programming problem, to find a maximum or minimum value of a given linear function under integer linear constraints. The method we use is the Branch-and-Bound algorithm.

The Branch-and-Bound algorithms proceed by dividing the original problem into two child problems successively, producing a tree-structured search space. If a certain node gives an actual integer solution (that is not necessarily optimal), and if other search nodes are guaranteed to have lower objective function values than that solution, then the latter nodes need not be searched. In this way, this method prunes sub-nodes through the search space to effectively cut down computation costs, but those costs still become quite high for large-scale problems, since the costs increase in an exponentially with the size of the problem.

As a parallelization of the Branch-and-Bound algorithm, we distribute the search nodes created through the branching process to different processors, and let these processors work on their own sub-problems sequentially. Each sequential search process communicates with other processes to prune the search nodes. Many search algorithms utilize heuristics to control the schedule of the order of the sub-nodes to be searched, thus reducing the number of nodes needed to obtain the final result. Therefore it is important, in parallel search algorithms, to balance the distributed load among processors, and to communicate information for pruning as quickly as possible between these processors. We adopted one of the best search heuristics used in sequential algorithms.

3.3.1 Formulation of Problems

We consider the following mixed-integer linear optimization problems.

Problem – *ILP*

Minimize the following objective function of real variables x_j and integer variables y_j,

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_j + \sum_{i=1}^{m} b_{ij} y_j \geq e_j \quad \text{for } 1 \leq j \leq l$$

$$\sum_{i=1}^{n} c_{ij} x_j + \sum_{i=1}^{m} d_{ij} y_j = f_j \quad \text{for } 1 \leq j \leq k$$

where

$$x_i \in \boldsymbol{R} \text{ and } x_i \geq 0 \text{ for } 1 \leq i \leq n$$

$$y_i \in \boldsymbol{Z} \text{ where } l_i \leq y_i \leq u_i \text{ and } l_i, u_i \in \boldsymbol{Z} \text{ for } 1 \leq i \leq m$$

a_{ij}, b_{ij}, c_{ij}, d_{ij}, e_i, f_i are real constants.

In practical situations integer variables y_j often take only 0, 1, but here we consider the general case.

3.3.2 Sequential Branch-and-Bound Algorithm

As a preparation to solve the above mixed-integer linear problems $\boldsymbol{ILP}$, we consider the continuously-relaxed problem $\boldsymbol{LP}$.

Problem – $\boldsymbol{LP}$
Minimize the following objective function of real variables x_j, y_j,

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_j + \sum_{i=1}^{m} b_{ij} y_j \geq e_j \quad \text{for } 1 \leq j \leq l$$
$$\sum_{i=1}^{n} c_{ij} x_j + \sum_{i=1}^{m} d_{ij} y_j = f_j \quad \text{for } 1 \leq j \leq k$$

where

$$x_i \in \boldsymbol{R} \text{ and } x_i \geq 0 \text{ for } 1 \leq i \leq n$$
$$y_i \in \boldsymbol{R} \text{ where } l_i \leq y_i \leq u_i \text{ and } l_i, u_i \in \boldsymbol{Z} \text{ for } 1 \leq i \leq m$$
$$a_{ij},\ b_{ij},\ c_{ij},\ d_{ij},\ e_i,\ f_i \text{ are real constants.}$$

$\boldsymbol{LP}$ can be solved by the simplex algorithm. If the values of original integer variables are exact integers, then it also gives the solution of $\boldsymbol{ILP}$. Otherwise, we take a non-integer value $\bar{y}_s$ for the solution of $\boldsymbol{LP}$, and impose two new interval constraints $\bar{y}_s$, $l_s \leq y_s \leq [\bar{y}_s]$ and $[\bar{y}_s]+1 \leq y_s \leq u_s$, where y_s is an integer variable, and obtain two child problems (Figure 9). Continuing this procedure, called branching, we continue to divide the search space to produce more constrained sub-problems as we proceed deeper into the tree structured search space. Eventually this process leads to a sub-problem having a continuous solution that is also an integer solution to the problem. Also we can select the best integer solution from those found in the process.

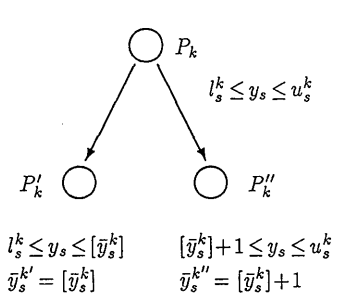


Figure 9: Branching of nodes

While the above branching process can only enumerate integer solutions, if we have a means of guaranteeing that a sub-problem cannot have a better solution than the already obtained integer solutions in terms of the optimum value of the objective function, then we can skip these sub-problems and need only search for the remaining nodes. For mixed-integer linear problems we can use the solutions for continuously relaxed problems as a criterion for pruning. Continuously relaxed problems always have a better optimum value for the objective function than the original integer problems. Sub-problems whose continuously relaxed problems have no better optimum than the already obtained integer solution cannot give a better optimum value, hence it becomes unnecessary to search further (bounding procedure).

Branch-and-Bound methods repeat branching and bounding in this way to obtain the final optimum. These sub-problems obtained through the branching process denote search nodes.

Sequential algorithm

Step 0 Initial setting
Let $\boldsymbol{ILP}_0$ mean the original problem $\boldsymbol{ILP}$, and $\mathcal{N}$ mean the set of search nodes. Set $\mathcal{N}$ to $\{\boldsymbol{ILP}_0\}$, and solve a continuously relaxed problem $\boldsymbol{LP}_0$. If an integer solution is obtained go to Step5. Otherwise set the incumbent solution $\tilde{z}$ to ∞ and go to Step1.

Step 1 Selecting branching node
If $\mathcal{N} = \emptyset$, then go to Step5.

If $\mathcal{N} \neq \emptyset$, then select the next branching node $\boldsymbol{ILP}_k$ out of $\mathcal{N}$ following the heuristics, and go to Step2.

Step 2 Selecting branching variable and branch
Select the integer variable y_s to be used for the branching process to work on $\boldsymbol{ILP}_k$ according to the heuristics, and branch with respect to it. Let the resulting two nodes be $\boldsymbol{ILP}_{k'}$, $\boldsymbol{ILP}_{k''}$
Go to Step3.

Step 3 Continuously relax two nodes
Solve two continuously relaxed problems $\boldsymbol{LP}_{k'}$ and $\boldsymbol{LP}_{k''}$ by the simplex algorithm. Go to Step4.

Step 4 Fathom two children nodes
If relaxed problem $\boldsymbol{LP}_{k'}$ does not have a solution, or gives a solution $\bar{z}_{k'}$ that is no better than the incumbent solution, in other words $\bar{z}_{k'} > \tilde{z}$, then stop searching (bounding operation).

If the point $(\bar{x}^{k'}, \bar{y}^{k'})$ to achieve a solution $\bar{z}_{k'}$ has integer value $\bar{y}$ and moreover gives a better solution than the incumbent solution obtained so far, in other words $\bar{z}_{k'} < \tilde{z}$, then let $\tilde{z} = \bar{z}_{k'}$, $\tilde{x} = \bar{x}^{k'}$ and $\tilde{y} = \bar{y}^{k'}$ (revision of the incumbent).

If $(\bar{x}^{k'}, \bar{y}^{k'})$ is not an integer solution and gives a better optimum value than the incumbent, then add this node, $\mathcal{N} := \mathcal{N} \cup \{ILP_{k'}\}$ (Addition of a node).

Do the same thing to $ILP_{k''}$, and go to Step1.

Step 5 End step

If $\tilde{z} \neq \infty$, then let the incumbent $(\tilde{x}, \tilde{y})$ be the optimum solution.

If $\tilde{z} = \infty$, then problem ILP has no solution.

3.3.3 Heuristics for Branching

The following two factors determine the schedule of the order in which the sequential search process goes through the nodes in the search space:

1. The priorities of sub-problems(nodes) to decide the next node on which the branching process operates.
2. Selection of a variable out of the integer variables with which the search space is divided.

It is preferable that the above selections are done in such a way that the actual nodes, searched in the process of finding the optimal, form as small a part as possible within the total search space. We adopted one of the best heuristics of this type from operations research as a basis of our parallel algorithm([Benichou *et al.* 71]).

Selection of sub-problems

We use a combination of depth-first strategy and best-first strategy(w.r.t. heuristic function). In each branching process, what is called the pseudo-costs $p_{up}(j)$, $p_{down}(j)$ of integer variables y_j are computed. These are the increase ratios of the optimum value of the continuously relaxed problem with regard to those integer variables. In the next heuristic function $h(ILP_k)$ of the node is calculated:
$h(ILP_k) = \tilde{z}_k + \sum_{j=1}^{n_2} \min\{p_{up}(j)(1-f_j), p_{down}(j)f_j\}$,
$f_j = \bar{y}_j^k - [\bar{y}_j^k]$,
Suppose the node ILP_k is divided into $ILP_{k'}$ and $ILP_{k''}$.

n-i. When at least one of these two nodes is not yet terminated, select the one having a better(i.e., smaller) heuristic value $h(ILP)$ as the next branching node (depth-first).

n-ii. When both have terminated,

a. if no incumbent solution has yet been found, select the latest node to which branching has been done (depth-first).

b. if an incumbent solution has already been found, select the node having the best heuristic function value (best-first).

Selection of the branching variable

To select the branching variable when trying to branch at the node ILP_k,

v-i. If no incumbent solution is found, select the variable y_j^k from those integer variables that do not take exact integer values in $(\bar{x}^k, \bar{y}^k)$, and which gives the greatest difference between the two increases in the heuristic value, namely the one to attain
$\max_j\{|p_{up}(j)(1-f_j) - p_{down}(j)f_j|; f_j\text{non-integer}\}$

v-ii. If an incumbent solution is found, select the variable y_j^k out of those integer variables that do not take exact integer values in $(\bar{x}^k, \bar{y}^k)$, and which gives the maximum of the minimum value of the left and right side heuristic values, namely that to attain
$\max_j\{\min\{p_{up}(j)(1-f_j), p_{down}(j)f_j; f_j\text{non-integer}\}$

3.3.4 Parallel Branch-and-Bound Method

The parallel algorithm derived from the above sequential algorithm is implemented on Multi-PSI. Our parallel algorithm exploits the independence of many sub-processes created through branching in the sequential algorithm, distributing these processes to different processors. What is necessary here is that the search space is divided as evenly as possible among processors to achieve good load balance, and that the pruning operation is performed by all the processors simultaneously. Also, incumbent solutions found in each processor need to be communicated between processors. The details of the parallel algorithm is described in the following.

Load balancing

One parent processor works on the sequential algorithm up to a certain depth d of the search tree. It then creates 2^d child nodes and distributes them to other processors as shown in Figure 10. These search nodes are allocated to different processors cyclically, where each of the processors works on these sub-problems sequentially. Therefore, load balancing is static in this case.

Distribution is done only at a certain depth of the search tree, to prevent the granularity of a node from being too small and to decrease the communication costs.

Heuristics for pruning

Each processor has a share of a certain number of sub-problems assigned, and works on these nodes with the same heuristics of branching node selection and branching variable selection as those of the sequential case. For the node selection heuristics, we use the priority control facility of KL1, to assign priorities to the search nodes on which the best-first strategy with the heuristic function can depend. (See [Oki *et al.* 89] for details of this technique.)

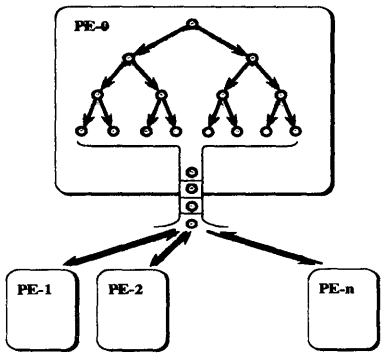


Figure 10: Generation of parallel processes

Transfer of global data

While the search space is distributed among different processors, if the information to prune nodes is not communicated well among them, then the processor has to work on unnecessary nodes, and the overall work becomes larger compared with the sequential version. This causes a reduction in the computation speed.

Therefore, incumbent solutions are transferred between processors to be shared so that each processor can update the current incumbent solution as soon as possible (Figure 11). This is realized by assigning a higher priority to the goal responsible for data transfer in the program.

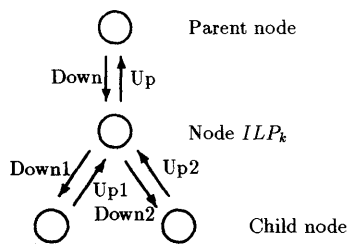


Figure 11: Report stream between nodes

3.3.5 Experimental Results

We implemented the above parallel algorithm in KL1, and experimented with job-shop scheduling problem. Table 6 shows a result of computation speedups for a 4job-3machine problem and the total number of searched nodes to get to the solution.

The situation often occurs where a processor visits an unnecessary node before the processor receives pruning information. This is because communication takes a time, and certainly cannot be instantaneous, in a distributed memory machine. Table 6 shows a case where this actually happens.

Table 6: Speedup

Processors	1	2	4	8
Speedup	1.0	1.5	1.9	2.3
Number of nodes	242	248	395	490

One of the problems in parallel search algorithms is how to decrease the growth of the size of the total search space compared with the sequential search algorithms.

4 GDCC Program Examples

Example 1 : integer programming

The following program is a simplified version of the integer programming used to find the integer solution that gives the minimum (or maximum) value of an objective function under given constraints. This program shows the basic structure of the Branch-and-Bound method.

```
:- module pseudo_integer_programming.
:- public integer_pro/3.

integer_pro(X,Y,Z):- true |
    call((simplex#X>=5,
          simplex#X+2*Y>=-3,
          simplex#X+Y-Z<=5)) initial nil giving Co,
    take_min(Co).

take_min(Co):- true |
    call(simplex#min(X+Y,Ans)) initial Co giving Co1,
    (Ans={minusinfinite,_} -> error;
     otherwise;
     Ans={_,[X=ValX|_]}    -> check(ValX,Co)).

check(ValX,Co):- kl1!integer(ValX) |
    solve_another_variables(Co).
otherwise.
check(ValX,Co):- true |
    floor(ValX,SupX,InfX),
    call(simplex#X=<InfX) initial Co giving Co1,
    take_min(Co1),
    call(simplex#X>=SupX) initial Co giving Co2,
    take_min(Co2).
```

The block in the clause `integer_pro` solves a set of constraints. The block in the clause `take_min` finds the minimum value of the given objective function. If the minimum value exists (not $-\infty$), `check` is called. In clause `check`, if the value of X, that gives the minimum value of the objective function is not an integer, two new constraints are added in order to the X become integer (for instance, if $X = 3.4$ then $X >= 4$ and $X <= 3$), and the minimum values with respect to the new constraints are solved again. Method `kl1!integer` decides whether the value X is an integer. Where, `kl1!` indicates KL1

method calling, a KL1 method is called from the GDCC program using this notation.

Synchronization between the inference engine and the solver to get the minimum value is achieved by the blocks in **integer_pro** and **take_min**. Multiple contexts are shown by the two blocks of **check**.

Example 2 : geometric problem

Next, we show how to use a function to find the approximated roots of uni-variate equations and how to handle multiple contexts using an example which is also used in [Aiba *et al.* 88].

```
:- module heron.
:- public tri/4, test1/4, test2/4.

tri(A,B,C,S) :- true |
    alloc(10,CA,CB,H),
    alg#C=CA+CB,
    alg#CA**2+H**2=A**2,
    alg#CB**2+H**2=B**2,
    alg#H*C=2*S.

test1(A,B,C,S) :- true |
    call( tri(A,B,C,S) ) initial nil giving GB,
    output1(GB).   % output to a window screen

test2(A,B,C,S) :- true |
    call( tri(A,B,C,S) ) initial nil giving GB,
    Err= 1/100000000,
    kl1!find:find(GB,Err,1,SubGB,UniEqs,UniSols),
    kl1!find:sol(SubGB,UniSols,Err,1,FGB),
    check(FGB, S).

check([], _) :- true | true.
check([FGB|FGBs], S) :- true |
    call( check_ask(S,Ans) ) initial FGB giving Sol,
    check_sub(Ans, Sol, FGBs, S).

check_sub(true, Sol, FGBs, S) :- true |
    output(Sol),  % output to a window screen
    check(FGBs, S).
check_sub(false, _, FGBs, S) :-  check(FGBs, S).

check_ask(S, Ans) :- alg#S > 0 | Ans = true.
check_ask(S, Ans) :- alg#S =< 0 | Ans = false.
```

Figure 12 shows the meaning of the constraints set contained in clause **tri**, where ** in equations indicates a power operation. **CA,CB,H** are local variables, **A, B, C** represents the three edges of a triangle, and **S** is its area. **alloc**(*Pre, Var1,..., VarN*) is a declaration to give precedence *Pre* to variables *Var1,..., VarN*. A monomial including a variable that has the highest **Pre** is the highest monomial, that is the precedence of variables is stronger than the degree in comparison.

If the goal,

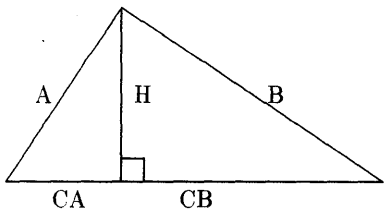


Figure 12: A triangle and its parameters

```
?- alloc(0,A,B,C),alloc(5,S),
   heron:test1(A,B,C,S).
```

is given, in which all parameters are free, this program outputs a Gröbner basis consisting of seven rules. Among them is the following rule that contains only **A, B, C** and **S**.

```
S**2= -1/16*C**4+1/8*C**2*B**2+1/8*C**2*A**2
      -1/16*B**4+1/8*B**2*A**2-1/16*A**4.
```

This is equivalent to Heron's formula. Of course, this program can be executed by a goal with concrete parameters. For example, when the goal,

```
?- alloc(5,S), heron:test1(3,4,5,S).
```

is given, the program produces **S**2= 36**.

However, the Buchberger algorithm cannot extract discrete values from this equation, as shown in section 3.1.2. Method **test2** approximates the real roots from a Gröbner basis, if the basis contains uni-variate equations. If the goal

```
?- alloc(5,S), heron:test2(3,4,5,S)
```

is given, first the constraint set is solved to obtain Gröbner basis **GB** using the **call** predicate, then uni-variate equations are extracted from **GB** using the method **find**:

```
kl1!find:find(GB,Err,1,SubGB,UniEqs,UniSols).
```

Where, **UniSols** contains the all combinations of solutions with precision **Err**, **UniEqs** is a set of the uni-variate equations extracted from Gröbner basis **GB**, and **SubGB** is the basis remaining after removing the uni-variate equations. The next method **sol** obtains a new Gröbner basis **FGB** by asserting the combinations of approximated solutions **UniSols** into **SubGB**. It is necessary to modify the Buchberger algorithm to handle approximated solutions, as explained in [Aiba *et al.* 91]. **FGB** contains plural Gröbner bases in list format, and these bases are filtered by the method **check**, which checks whether $S > 0$ is satisfied at the guard of the sub-block **check_ask**.

5 Conclusion

GDCC is an instance of the cc language and satisfies two levels of parallelism: the execution of an inference engine and solvers in parallel, and the execution of a solver in parallel. A characteristic of a cc language is that it is more declarative than sequential CLP languages, since the guard part is the only synchronization point between an inference engine and solvers. GDCC inherits this characteristic and, moreover, it has a block mechanism to synchronize meta-operations with constraints.

In the latest (master-slave) version of the parallel algebraic solver, the parallel execution of "cyclic 6-roots" with 16 processors is 4.42 times faster than execution with a single processor. With the boolean solver, parallel execution of the 6 queens problem with 16 processor is 2.28 times faster than with a single processor. We also show the realization of fast parallel search for mixed integer programming using the Branch-and-Bound algorithm.

The following items are yet to be studied. As shown in the program examples, current users must describe everything explicitly to handle multiple contexts. Thus, support faculties and utilities to handle multiple contexts are required. We will also improve the parallel constraint solvers to obtain both good absolute performance and better parallel speedup. The algebraic solver requires parallel speedup. The boolean solver needs to increase the parallel executable parts of its algorithm. The linear integer solver has to improve the ratio of pruning in parallel execution. Through these refinements and experiments using the handling robot design system, we can realize a parallel CLP language system that has high functionality in both its language facilities and performance.

6 Acknowledgments

We would like to thank Professor Makoto Amamiya at Kyushu University and the members of the CLP working group for their discussions and suggestions. We would also like to thank Dr. Fuchi, Director of the ICOT Research Center, and Dr. Hasegawa, Chief of the Fourth and Fifth Laboratory, for their encouragement and support in this work.

References

[Aiba *et al.* 88] A. Aiba, K. Sakai, Y. Sato, D. Hawley and R. Hasegawa. Constraint Logic Programming Language CAL. In *International Conference on Fifth Generation Computer Systems*, pages 263–276, 1988.

[Aiba *et al.* 91] A. Aiba, S. Sato, S. Terasaki, M. Sakata and K. Machida. CAL: A Constraint Logic Programming Language – Its Enhancement for Application to Handling Robots –. Technical Report TR-729, Institute for New Generation Computer Technology, 1991.

[Backelin and Fröberg 91] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor, *Proc. ISSAC'91* pages 103–111, ACM, July 1991.

[Benichou *et al.* 71] M. Benichou, L. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent. Experiments in Mixed-Integer Linear Programming. In *Mathematical Programming* 1 pages 76–94, 1971.

[Boege *et al.* 86] W. Boege, R. Gebauer and H. Kredel. Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases. *Symbolic Computation*, 2(1):83–98, 1986.

[Buchberger 83] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.

[Buchberger 85] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[Cangilia *et al.* 88] L. Caniglia, A. Galligo and J. Heintz. Some new effectivity bounds in computational geometry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes - 6th International Conference*, pages 131–151. Springer-Verlag, 1988. Lecture Notes in Computer Science 357.

[Chikayama *et al.* 88] T. Chikayama, H. Sato and T. Miyazaki. Overview of Parallel Inference Machine Operationg System (PIMOS). In *International Conference on Fifth Generation Computer Systems*, pages 230–251, 1988.

[Clarke *et al.* 90] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, October 1990.

[Colmerauer 87] A. Colmerauer. Opening the Prolog III Universe: A new generation of Prolog promises some powerful capabilities. *BYTE*, pages 177–182, August 1987.

[Dincbas *et al.* 88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Bertheir. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[Gebauer and Möller 88] R. Gebauer and H. M. Möller. On an installation of Buchberger's algorithm. *Symbolic Computation*, 6:275–286, 1988.

[Hawley 91a] D. J. Hawley. A Buchberger Algorithm for Distributed Memory Multi-Processors. In *The first International Conference of the Austrian Center for Parallel Computation*, Salzburg, September, 1991. Also in Technical Report TR-677 Institute for New Generation Computer Technology, 1991.

[Hawley 91b] D. J. Hawley. The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver. Technical Report TR-713 Institute for New Generation Computer Technology, 1991.

[Hentenryck 89] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Prelimiary Results of CHIP within PEPSys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Jaffar and Lassez 87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *4th IEEE Symposium on Logic Programming*, 1987.

[Li 86] G.-J. Li and W. W. Benjamin. Coping with Anomalies in Parallel Branch-and-Bound Algorithms, *IEEE Trans. on Computers*, 35(6): 568–573, June 1986.

[Maher 87] M. J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[Oki *et al.* 89] H. Oki, K. Taki, S. Sei, and M. Furuichi. Implementation and evaluation of parallel Tsumego program on the Multi-PSI. In *Proceedings of the Joint Parallel Processing Symposium (JSSP'89)*, pages 351–357, 1989. (In Japanese).

[Ponder 90] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. D. Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 51–74, Academic Press, 1990.

[Quinn 90] M. J. Quinn. Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multiprocessor. *IEEE Trans. on Computers*, 39(3):384–387, March 1990.

[Sakai and Aiba 89] K. Sakai and A. Aiba. CAL: A Theoritical Background of Constraint Logic Programming and its Applications. *Symbolic Computation*, 8(6):589–603, 1989.

[Saraswat 89] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[K. Satoh and Aiba 90] K. Satoh and A. Aiba. Hierarchical Constraint Logic Language: CHAL. Technical Report TR-592, Institute for New Generation Computer Technology, 1990.

[K. Satoh 90b] K. Satoh. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, Institute for New Generation Computer Technology, 1990.

[S. Sato and Aiba 90] S. Sato and A. Aiba. An Application of CAL to Robotics. Technical Memorandum TM-1032, Institute for New Generation Computer Technology, 1990.

[Y. Sato and Sakai 88] Y. Sato and K. Sakai. Boolean Gröbner Base, February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[Senechaud 90] P. Senechaud. Implementation of a Parallel Algorithm to Compute a Gröbner Basis on Boolean Polynomials. In J. D. Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[Siegl 90] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[Takeda *et al.* 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *International Conference on Fifth Generation Computer Systems*, pages 978–986, 1988.

[Ueda and Chikayama 90] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33(6):494–500, December 1990.

[Vidal 90] J. P. Vidal. The Computation of Gröbner Bases on a Shared Memory Multi-processor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

cu-Prolog for Constraint-Based Grammar

Hiroshi TSUDA
Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
E-mail : tsuda@icot.or.jp

Abstract

cu-Prolog is a constraint logic programming (CLP) language appropriate for natural language processing such as a Japanese parser based on JPSG. Compared to other CLP languages, cu-Prolog has several unique features. Most CLP languages take algebraic equations or inequations as constraints. cu-Prolog, on the other hand, takes the Prolog atomic formulas of user-defined predicates. cu-Prolog, thus, can describe symbolic and combinatorial constraints that are required for constraint-based natural language grammar description. As a constraint solver, cu-Prolog uses unfold/fold transformation dynamically with some heuristics.

JPSG (Japanese Phrase Structure Grammar) is a constraint-based and unification-based Japanese grammar formalism beging developed by the PSG-working group at ICOT. Like HPSG (Head-driven Phrase Structure Grammar), JPSG is a phrase structure whose nodes are feature structures. Its grammar description is mainly formalized by local constraints in phrase structures.

This paper outlines cu-Prolog and its application to the disjunctive feature structure and JPSG parser.

1 Introduction

Two aspects are considered to classify contemporary natural language grammatical theories[Carpenter *et al.* 91]. Firstly, They must be classified according to whether they have transformation operations among different structure levels.

One current version of *transformational grammar* is GB (Government and Binding) theory[Chomsky 81]. So called *unification-based grammars*[Shieber 86], such as GPSG (Generalized Phrase Structure Grammar), LFG (Lexical Functional Grammar), HPSG (Head-driven Phrase Structure Grammar)[Pollard and Sag 87], and JPSG (Japanese Phrase Structure Grammar)[Gunji 86] are categorized as *non-transformational grammars*. Unification-based grammar is a phrase structure grammar whose nodes are feature structures. It uses unification as its basic operation. In this respect, it is congenial to logic programming.

Secondly, classification must be made as to whether a language's grammar description is *rule-based* or *constraint-based*[1]. GPSG and LFG fall into the former category. The latter includes GB theory, HPSG, and JPSG. From the viewpoint of procedural computation, rule-based approaches are better. However, by constraint-based approaches, more general and richer grammar formalisms are possible because morphology, syntax, semantics, and pragmatics are all uniformly treated as constraints. Also, the most important feature of constraints, the declarative grammar description, allows various information flows during processing.

Consider the programming languages used to implement these grammatical theories. For rule-based grammars, many approaches have been attempted, such as FUG[Kay 85] and PATR-II[Shieber 86]. As yet, however, no leading work has been done on constraint-based grammars.

Our constraint logic programming language *cu-Prolog* [Tsuda *et al.* 89b, Tsuda *et al.* 89a] aims to provide an implementation framework for constraint-based grammars. Unlike most CLP languages, cu-Prolog takes the Prolog atomic formulas of user-defined predicates as constraints.

cu-Prolog originated from the technique of *constrained unification* (or *conditioned unification* [Hasida and Sirai 86]) - a unification between two constrained Prolog patterns. The basic component of cu-Prolog is a *Constrained Horn Clause (CHC)* that adds constraints in terms of user-defined Prolog predicates to Horn clauses. Their domain is suitable for symbolic and combinatorial linguistic constraints. The constraint solver of cu-Prolog uses the unfold/fold [Tamaki and Sato 83] transformation dynamically with certain heuristics.

This paper illustrates

- the outline of cu-Prolog,
- treatment of disjunctive feature structures with PST(Partially Specified Term)[Mukai 88] in cu-Prolog, and

[1]Constraint-based approaches are also called *information-based* or *principle-based* approaches.

- the JPSG parser as its most successful application.

2 Linguistic Constructions

As an introduction, this section explains the various types of linguistic constraints in constraint-based grammar formalisms.

2.1 Disjunctive Feature Structure

Unification-based grammars utilize *feature structures* as basic information structures. A feature structure consists of a set of pairs of labels and their values. In (1), *pos* and *sc* are called *features* and their values are n and a singleton set $< [pos = p] >$.

$$\left[\begin{array}{l} pos = n \\ sc = \left\langle \left[pos = p \right] \right\rangle \end{array}\right] \quad (1)$$

Morphological, syntactic, semantic, and pragmatic information are all uniformly stored in a feature structure.

Moreover, natural language descriptions essentially require some framework to handle ambiguities such as polysemic words, homonyms, and so on. *Disjunctive feature structures* are widely used to handle disjunctions in feature structures[Kay 85]. Disjunctive feature structures consist of the following two types.

Value disjunction A value disjunction specifies the alternative values of a single feature. The following example states that the value of the *pos* feature is n or v, and the value of the *sc* feature is $<>$ (empty set) or $< [pos = p] >$.

$$\left[\begin{array}{l} pos = \{n, v\} \\ sc = \left\{ \begin{array}{l} \langle\rangle \\ \left\langle \left[pos = p \right] \right\rangle \end{array} \right\} \end{array}\right] \quad (2)$$

General disjunction A general disjunction specifies alternative groups of multiple features. In the following structure, $sem = love(X, Y)$ is common, and the rest is ambiguous.

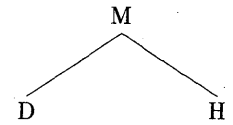
$$\left[\begin{array}{l} \left\{ \begin{array}{l} \left[pos = n \right] \\ \left[\begin{array}{l} pos = v \\ vform = vs \\ sc = \left\langle \left[pos = p \right] \right\rangle \end{array}\right] \end{array} \right\} \\ sem = love(X, Y) \end{array}\right] \quad (3)$$

One serious problem in treating disjunctive feature structures is the computational complexity of their unification problem because it is essentially NP-complete[Kasper and Rounds 86]. Some practically efficient algorithms to deal with disjunctions have been studied by [Kasper 87] and [Eisele and Dörre 88].

2.2 Structural Principles

Unification-based grammars are phrase structures whose nodes are feature structures. Their grammar descriptions consist of both phrase structure rules and local constraints in a phrase structure. In current unification-based grammars, such as HPSG and JPSG, phrase structure rules become very general and grammars are mainly described with a set of local constraints called *structural principles*.

JPSG has only one phrase structure rule, as follows.



M, D and H are the *mother*, the *dependent* daughter, and the *head* daughter respectively. This phrase structure is applicable to both the *complementation structure* and *adjunction structure* of Japanese[2]. In complementation structures, D acts as a complement. In adjunction structures, D works as a modifier.

Structural principles are relations between the features of three nodes (M, D and H) in a local tree. In the following, we explain some features and their constraints.

mod: The *mod* feature specifies the function of D in a phrase structure. When the value is +, D works as a modifier, and when -, it works as a complement.

head features: Features such as *pos*, *gr*, *case*, and *infl* are called *head features*. These conform to the following *head feature principle*.

> The value of a head feature of M unifies with that of H.

subcat features: Features *subcat* and *adjacent* are called *subcat features*. They take a set of feature structures that specify adjacent categories such as complements, and nouns. The *subcat feature principle* is

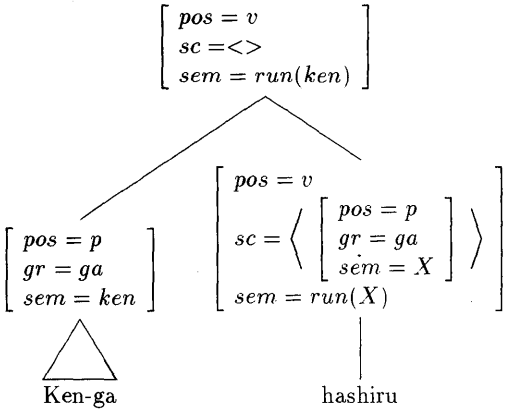
> In the complementation structure, the value of a subcat feature of M unifies with that of H minus D. In the adjunction structure, the value of a subcat feature of M unifies with that of H.

sem: The *sem* feature specifies semantic information.

> In the complementation structure, the *sem* value of M unifies with that of H. In the adjunction structure, the *sem* value of M unifies with that of D.

[2]For example, "Ken-ga aisuru (Ken loves)" is the complementation structure, and "ooki-na yama (big mountain)" is the adjunction structure.

Below is the analysis for "Ken-ga hashiru (Ken runs)."



3 cu-Prolog

3.1 Conventional Approaches

Prolog is often used as an implementation language for unification-based grammars. However, its execution strategy is fixed and procedural, i.e., always from left to right for AND processes, and from top to bottom for OR processes. Prolog programmers have to align goals such that they are solved efficiently. Prolog, therefore, is not well-suited for constraint-based grammars because it is impossible to stipulate in advance which type of linguistic constraints are to be processed in what order.

Some Prolog-like systems such as PrologII and CIL[Mukai 88] have bind-hook mechanisms that can delay some goals (constraints) until certain variables bind. As the mechanism, however, can only check constraints by executing them, it is not always efficient.

Most CLP languages, such as CLP(R)[Jaffar and Lassez 87], PrologIII, and CAL, take the constraints of algebraic domain with equations or inequations. Their constraint solvers are based on algebraic algorithms such as Gröbner bases, and solving equations. However, for AI applications and especially natural language processing systems, symbolic and combinatorial constraints are far more desirable than algebraic ones. cu-Prolog, on the other hand, can use symbolic and combinatorial constraints because its constraint domain is the Herbrand universe.

3.2 Constrained Horn Clause (CHC)

The basic component of cu-Prolog is the *Constrained Horn Clause (CHC)*[3].

[Def] 1 (CHC) *The* Constrained Horn Clause (CHC) *is*

$$\overbrace{HEAD}^{Head} : - \overbrace{B_1, B_2, \ldots, B_n}^{Body}; \overbrace{C_1, C_2, \ldots, C_m}^{Constraint}.$$

$HEAD$, called **head**, *is an atomic formula, and $B_1, \ldots, B_n$, called* **body**, *is a sequence of atomic formulas. $C_1, \ldots, C_m$, called* **constraint**, *is a sequence of atomic formulas or equal constraints of the form: $Variable = Term$. Body or constraint can be empty.* □

From the viewpoint of declarative semantics, the above clause is equivalent to the following Horn Clause.

$$HEAD \ : - \ B_1, B_2, \ldots, B_n, C_1, C_2, \ldots, C_m .$$

3.3 Derivation Rule

cu-Prolog expands the derivation rule of Prolog by adding a constraint transformation operation.

$$\frac{\overbrace{A.\mathbf{K}:\mathbf{C}.}^{goal} \quad \overbrace{A' : -\mathbf{L}:\mathbf{D}.}^{program} \quad \overbrace{\theta = mgu(A, A')}^{substitution} \quad \overbrace{\mathbf{C}' = mf(\mathbf{C}\theta + \mathbf{D}\theta)}^{constraint\ transformation}}{\underbrace{\mathbf{L}\theta.\mathbf{K}\theta:\mathbf{C}'.}_{new\ goal}}$$

A and A' are heads. $\mathbf{K}$ and $\mathbf{L}$ are bodies. $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{C}'$ are constraints. $mgu(A, A')$ is the most general unifier between A and A'. $mf(Cstr)$ is a canonical form of a constraint that is equivalent to $Cstr$.

As a computational rule, when the transformation of $\mathbf{C}\theta + \mathbf{D}\theta$ fails, the above derivation rule is not applied.

3.4 PST

cu-Prolog adopts PST (Partially Specified Term) [Mukai 88] as a data structure that corresponds to the feature structure in unification-based grammars.

[Def] 2 (Partially Specified Term (PST)) PST *is a term of the following form :*

$$\{l_1/t_1, l_2/t_2, \ldots, l_n/t_n\} .$$

l_i, called **label**, *is an atom and $l_i \neq l_j (i \neq j)$. t_i, called* **value**, *is a term.* □

A recursive PST structure is not allowed.

[Def] 3 (constrained PST) *In cu-Prolog, PST is stored as an equal constraint with other relevant constraints :*

$$\mathtt{X} = \mathtt{PST}.\, \mathtt{c_1(X)}, \mathtt{c_2(X)}, \ldots, \mathtt{c_n(X)}$$

We call the above type of constraints constrained PST. `X=PST` *corresponds to [Kasper 87]'s unconditional conjunct, and* $\mathtt{c_1(X)}, \mathtt{c_2(X)}, \ldots, \mathtt{c_n(X)}$ *corresponds to the conditional conjunct.* □

In the next subsection, we give its canonical form *modular*. The constrained PST can naturally describe disjunctive feature structures of unification based grammars.

[3] Or *Constraint Added Horn Clause (CAHC).*

3.5 Canonical form of a constraint

The canonical form of a constraint in CHC is called *modular*. First, we give an intuitive definition of modular without PST.

[Def] 4 (modular (without PST)) *A sequence of atomic formulas $C_1, \ldots, C_m (m > 1)$ is* **modular** *when all its arguments are different variables.*

For example,

`member(X,Y),member(U,V)` is modular,
`member(X,Y),member(Y,Z)` is not modular, and
`append(X,Y,[a,b,c,d])` is not modular.

We expand the definition of *modular* for constrained PST.

[Def] 5 (component) *The* **component** *of an argument of a predicate is a set of labels to which the argument may bind. Here, an atom or a complex term is regarded as a PST of the label* `[]`. □

`Cmp(p,n)` stands for the component of the nth argument of a predicate `p`. `Cmp(T)` represents a set of labels of a PST `T`. In a constraint of the form `X=t`, variable `X` is regarded as taking `Cmp(t)`.

Components can be computed by static analysis of the program [Tsuda 91]. *Vacuous argument places*[Tsuda and Hasida 90] are arguments whose components are ϕ.

Consider the following example.

```
c0({f/b},X,Y):-c1(Y,X).
c0(X,b,_):-X={g/c},c2(X).
c1(X,X).
c1(X,[X|_]).
c2({h/a}).
c2({f/c}).
```

The components are computed as follows.

```
Cmp(c0,1)={f,g,h}
Cmp(c0,2)=Cmp(c1,2)={[]}
Cmp(c0,3)=Cmp(c1,1)={}
Cmp(c2,1)={f,h}
```

[Def] 6 (dependency) *A constraint has* **dependency** *when*

1. *a variable occurs in two distinct places where their components have common labels.*
2. *a variable occurs in two distinct places where one component is* `{[]}` *and another component does not contain* `[]`, *or*
3. *the binding of an argument whose component is not ϕ.* □

[Def] 7 (modular (with PST)) *A constraint is* **modular** *when it contains no dependency. A Horn clause is* **modular** *when its body has no dependency.* □

User-defined predicates in a constraint must be defined with modular Horn clauses [4].

3.6 Constraint Transformation

The constraint solver ($mf(Cstr)$) transforms non-modular constraints into modular ones by deriving new predicates. In the following, we refer to this solver as the *constraint transformer*. The constraint transformer uses the *unfold/fold transformation* dynamically. [Tamaki and Sato 83]

3.6.1 Unfold/fold transformation

Let $\mathcal{T}$ be a set of program Horn clauses, Σ be initial constraints $\{C_1, \ldots, C_n\}$ that contain variables $x_1, \ldots, x_m$, and p be a new m-ary predicate.

Let $\mathcal{P}_i$ and $\mathcal{D}_i$ be sequences of sets of clauses that are initially defined as follows.

$$\begin{aligned} \mathcal{D}_0 &= \{p(x_1, \ldots, x_m) : -C_1, \ldots, C_n.\} \\ \mathcal{P}_0 &= \mathcal{T} \cup \mathcal{D}_0 \end{aligned}$$

$mf(\Sigma)$ returns $p(x_1, \ldots, x_m)$, if and only if there exists a sequence of program Horn clauses

$$\mathcal{P}_0, \ldots, \mathcal{P}_l$$

and every clause in $\mathcal{P}_l$ is modular.

$\mathcal{P}_{i+1}$ and $\mathcal{D}_{i+1}$ are derived from $\mathcal{P}_i$ and $\mathcal{D}_i$ by one of the following three types of transformations ($0 \leq i < l$).

1. unfolding

$$\frac{\begin{array}{c}\mathcal{P}_i = \{H : -A, \mathbf{R}\} \cup \mathcal{P}_i' \\ A_j : -\mathbf{B}_j \in \mathcal{P}_i, \quad A_j\theta_j = A\theta_j \quad (1 \leq j \leq m)\end{array}}{\begin{array}{c}\mathcal{P}_{i+1} = \bigcup_{j=1}^{m} H\theta_j : -\mathbf{B}_j\theta_j, \mathbf{R}\theta_j \cup \mathcal{P}_i' \\ \mathcal{D}_{i+1} = \mathcal{D}_i\end{array}}$$

Here, A, A_j are atomic formulas and $\mathbf{R}, \mathbf{B}_j$ are sequences of atomic formulas ($1 \leq j \leq m$).

2. folding

$$\frac{\begin{array}{c}\mathcal{P}_i = \{H : -\mathbf{C}, \mathbf{R}\} \cup \mathcal{P}_i' \\ A : -\mathbf{B} \in \mathcal{D}_i. \quad \mathbf{B}\theta = \mathbf{C}\end{array}}{\begin{array}{c}\mathcal{P}_{i+1} = H : -A\theta, \mathbf{R} \cup \mathcal{P}_i' \\ \mathcal{D}_{i+1} = \mathcal{D}_i\end{array}}$$

Here. $\mathbf{C}$ and $\mathbf{R}$ have no common variables.

[4] For example, `member/2`, `append/3`, and finite predicates are defined with modular Horn clauses.

3. **definition**
Let **B** be a sequence of atomic formulas, $x_1, \ldots, x_n$ be variables in **B**, and p be a new predicate.

$$\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{p(x_1, \ldots, x_n) : -\mathbf{B}.\}$$
$$\mathcal{P}_{i+1} = \mathcal{P}_i$$

3.6.2 Example of Constraint Transformation

The following example shows a transformation of `member(A,Z),append(X,Y,Z)`.
$\mathcal{T}$ = { `T1,T2,T3,T4` }, where

```
T1 = member(X,[X|Y]).
T2 = member(X,[Y|Z]):-member(X,Z).
T3 = append([],X,X).
T4 = append([A|X],Y,[A|Z]):-append(X,Y,Z).
```

and

$$\Sigma = \{\texttt{member(A,Z)}, \texttt{append(X,Y,Z)}\}$$

With a new predicate `p1/4` derived as `D1`,

```
D1 = p1(A,X,Y,Z):-member(A,Z),append(X,Y,Z).
```

we get

$$\mathcal{D}_0 = \{\texttt{D1}\}\ \mathcal{P}_0 = \mathcal{T} \cup \{\texttt{D1}\}$$

Step 1: By unfolding of the first formula of `D1`'s body (`member(A,Z)`), we get

```
T5 = p1(A,X,Y,[A|Z]):-append(X,Y,[A|Z]).
T6 = p1(A,X,Y,[B|Z]):-member(A,Z),append(X,Y,[B|Z]).
```

$$\mathcal{P}_1 = \mathcal{T} \cup \{\texttt{T5}, \texttt{T6}\}$$

Step 2: By defining new predicates `p2/4` and `p3/5` as `D2,D3`,

```
T5' = p1(A,X,Y,[A|Z]):-p2(X,Y,A,Z).
T6' = p1(A,X,Y,[B|Z]):-p3(A,Z,X,Y,B).
D2  = p2(X,Y,A,Z):-append(X,Y,[A|Z]).
D3  = p3(A,Z,X,Y,B):-member(A,Z),append(X,Y,[B|Z]).
```

we get

$$\mathcal{D}_2 = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}\ \mathcal{P}_2 = \mathcal{T} \cup \{\texttt{T5}', \texttt{T6}', \texttt{D2}, \texttt{D3}\}$$

Step 3: By unfolding `D2`,

```
T7 = p2([],[A|Z],A,Z).
T8 = p2([B|X],Y,A,Z):-append(X,Y,Z).
```

$$\mathcal{P}_3 = \mathcal{T} \cup \{\texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{D3}\}$$

Step 4: Unfolding the second formula of `D3`'s body (`append(X,Y,[B|Z])`) gives

```
T9  = p3(A,Z,[],[B|Z],B):-member(A,Z).
T10 = p3(A,Z,[B|X],Y,B):-member(A,Z),append(X,Y,Z).
```

$$\mathcal{P}_4 = \mathcal{T} \cup \{\texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}\}.$$

Step 5: Folding `T10` by `D1` generates

```
T10' = p3(A,Z,[B|X],Y,B):-p1(A,X,Y,Z).
```

Accordingly,

$$\mathcal{P}_5 = \mathcal{T} \cup \{\texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}'\}.$$

Every clause in $\mathcal{P}_5$ is modular. As a result, `member(A,Z),append(X,Y,Z)` has been transformed into `p1(A,X,Y,Z)`, preserving equivalence, and new predicates `p1/4`,`p2/4`, and `p3/5` have been defined with `T5'`,`T6'`,`T7`,`T8`,`T9`, and `T10'`.

3.6.3 Example of constrained PST unification

Unification between constrained PSTs is done with PST unification followed by the transformation of relevant constraints.

The following example from [Eisele and Dörre 88] shows unification between two disjunctive feature structures:

$$\left[\, \mathtt{a} = \left\{ \left[\begin{array}{l} \mathtt{b} = + \\ \mathtt{c} = - \end{array}\right], \left[\begin{array}{l} \mathtt{b} = - \\ \mathtt{c} = + \end{array}\right] \right\} \,\right] \quad \text{and} \quad \left[\begin{array}{l} \mathtt{a} = \left[\, \mathtt{b} = \mathtt{V} \,\right] \\ \mathtt{d} = \mathtt{V} \end{array}\right]$$

These disjunctive feature structures are encoded in the two constrained PSTs. `X={a/U},s(U)` and `Y={a/{b/V},d/V}`, where

```
s({b/+,c/-}).    % definition of s/1
s({b/-,c/+}).
```

PST unification between `X` and `Y` gives

```
X=Y={a/U,d/V},U={b/V},s(U).
```

There is a dependency in terms of a label `b` because `Cmp(s,1)={b,c}`.

By defining a new predicate `c1/2`. `U={b/V},s(U)` becomes equivalent to `U={b/V},c1(U,V)`. Here, `c1/2` is defined as follows. [5]

```
c1({c/-},+).
c1({c/+},-).
```

Note that `X=Y={a/U,d/V},U={b/V},c1(U,V)` does not have any dependency because `Cmp(c1,1)={c}`.

[5] Precisely, *abstraction* operation in [Tsuda 91] is used in this transformation. In *abstraction*, PST unifications are made in terms of relevant labels alone for efficiency.

4 JPSG parser in cu-Prolog

JPSG (Japanese Phrase Structure Grammar)[Gunji 86] is a constraint-based and unification-based grammar designed specifically for Japanese. It is being developed by the PSG working group at ICOT.

To implement unification-based grammars, we have to consider how to describe and process feature structures for the first time. In cu-Prolog, PST enables the natural implementation of non-disjunctive feature structures. The labels of PST correspond to the features of a feature structure. As mentioned earlier, disjunctive feature structures correspond to constained PSTs.

In cu-Prolog, both disjunctive feature structures and structural principles are treated as constarints in CHC.

4.1 Encoding Lexical Ambiguity

As an example of the disjunctive feature structures, this subsection explains lexical ambiguities in this subsection. Consider the lexicons of homonyms or polysemic words. If the lexicon of an ambiguous word is divided into plural entries in terms of its ambiguity, the parsing process may be inefficient in that it sometimes backtracks to consult the lexicon. In constraint-based natural language processing, such ambiguity is packed as a constraint in a lexicon.

Below is a sample lexicon of Japanese auxiliary verb "reru." "reru" follows a verb whose inflection type is `vs` or `vs1`. If the adjacent verb is transitive, "reru" indicates plain passive. If the verb is intransitive, "reru" indicates affective passive [6]. These combinations are represented by adding constraints of `reru_form/1` and `reru_sem/4` in one lexical entry.

```
%% lexical entry of ''reru''
lex(reru,{sc/SC, sem/Sem,
       adjacent/{pos/v,infl/I,sc/VSC,sem/Sem}});
   reru_form(I),     % inflection (constraint)
   reru_sem(VSC,Vsem,SC,Sem).
     % combination of subcat and sem (constraint)

%%%%%% definition of constraints %%%%%%
reru_form(vs). % inflection type of the adjacent verb
refu_form(vs1).

% constraint for intransitive (affective) passive
reru_sem([{form/ga,sem/Sbj}],Sem,
       [{form/ga,sem/A},{form/ni,sem/Sbj}],
       affected(A,Sem)).
% constraint for transitive (normal) passive
reru_sem([{form/ga,sem/Sbj},{form/wo,sem/Obj}],
       Sem,
       [{form/ga,sem/Obj},{form/ni,sem/Sbj}],
       Sem).
```

[6] For example, "Ken ga ame ni fu-ra-reru" (Ken is affected by the rain.)

This lexicon is a representation of the following disjunctive feature structure.

$$\left[\left\{\begin{array}{l}\left[\begin{array}{l} adjc = \left[\begin{array}{l} sc = \left\langle \left[\begin{array}{l} pos = ga \\ sem = S1 \end{array}\right] \right\rangle \\ sem = Sem1 \end{array}\right] \\ sc = \left\langle \left[\begin{array}{l} form = ga \\ sem = A \end{array}\right], \left[\begin{array}{l} form = ni \\ sem = S1 \end{array}\right] \right\rangle \\ sem = affected(A, Sem1) \end{array}\right] \\ \left[\begin{array}{l} adjc = \left[\begin{array}{l} sc = \left\langle \left[\begin{array}{l} pos = ga \\ sem = S2 \end{array}\right], \left[\begin{array}{l} pos = wo \\ sem = O2 \end{array}\right] \right\rangle \\ sem = Sem2 \end{array}\right] \\ sc = \left\langle \left[\begin{array}{l} pos = ga \\ sem = O2 \end{array}\right], \left[\begin{array}{l} pos = ni \\ sem = S2 \end{array}\right] \right\rangle \\ sem = Sem2 \end{array}\right] \end{array}\right\} \right.$$
$$\left. adjc = \left[\begin{array}{l} pos = v \\ infl = \{vs1, vs2\} \end{array}\right] \right]$$

Although the lexicon is ambiguous, however, many kinds of constraints are automatically accumulated for solving during parsing. The disambiguation process in parsing is naturally realized by the constraint transformation of cu-Prolog. It has no need to write any special procedure for disambiguation.

4.2 Encoding Structural Principle

As mentioned in Section 2, the structural principles of JPSG are relations among features of three categories in a local tree. Intuitively, structure principles are encoded as constraints to a phrase structure rule:

$$psr(M, D, H); sp_1(M, D, H), \ldots, sp_n(M, D, H).$$

Here, $psr/3$ is a phrase structure rule and each $sp_i/3$ is a structure principle.

In cu-Prolog, these structural principles are evaluated flexibly with heuristics. In Prolog, however, above phrase structure rule is represented as

$$psr(M, D, H) :- sp_1(M, D, H), \ldots, sp_n(M, D, H).$$

Each principle is always evaluated sequentially. Prolog, therefore, is not well-suited for constraint based grammars because it is impossible to stipulate in advance which kind of linguistic constraints must be processed in what order.

As the first example, the principle of the *sem* feature in Section 2 is encoded as a constraint `sfp(M,D,H)`, where

```
sfp({sem/HS},{mod/+},{sem/HS}).
sfp({sem/HS},{mod/+},{sem/HS}).
```

As the second example, the *Foot Feature Principle* is defined as follows[Gunji 86].

> The value of FOOT feature of the mother unifies with the union of those of her daughters.

It is represented as constraint `ffp(M,D,H)`, where

```
ffp({foot/MF},{foot/DF},{foot/HF}) :-
   union(DF,HF,MF).
```

5 Implementation

cu-Prolog has been implemented in the C language of UNIX4.2/3BSD and the Apple Macintosh[Sirai 91]. cu-PrologIII [Tsuda *et al.* 92] is the latest implementation.

This section presents some implementation issues that relate particularly to the constraint transformer.

5.1 Constraint Transformer

5.1.1 Constraint Transformation Strategy

cu-Prolog uses the following three clause pools during constraint transformation.

DEFINITION: derivation clauses of new predicates

NON-MODULAR: non-modular clauses

MODULAR: modular clauses

The following is the transformation procedure of cu-Prolog.

1. If **DEFINITION** is not empty, remove one clause from **DEFINITION** and unfold it.
2. If **DEFINITION** is empty but **NON-MODULAR** is not empty, remove one clause N from **NON-MODULAR**. If N's head is modular, unfold N. If not, fold N or derive new predicates to N's body.
3. Repeat the above operations until **DEFINITION** and **NON-MODULAR** are both empty.

5.1.2 Heuristics

One of the outstanding features of cu-Prolog is the heuristics used in the constraint transformation.

The following three choices are available.

- selection of a clause from DEFINITION
- selection of a clause from NON-MODULAR
- selection of a formula to unfold

DEFINITION and **NON-MODULAR** are implemented by stacks, that is, the constraint transformer selects the latest. In unfolding, the *activation value* of each atomic formula is computed from the following formulas and the atomic formula of the highest value is unfolded.

$Arity$ = arity of the formula
$Const$ = number of arguments that bind to constants
$Vnum$ = total number of occurrence of variables in the formula
$Funct$ = number of arguments that bind to complex terms
Rec = If the predicate is recursively defined then 1, otherwise 0
$Defs$ = number of definition clauses of the predicate
$Units$ = number of unit clauses in the predicate definition
$Facts$ = If $Defs = Units$ then 1, otherwise 0

The activation value A of an atomic formula is computed using the following formula.

$$A = 3 * Const + 2 * Funct + Vnum - Defs + Units - 2 * Rec + 3 * Facts$$

We define each factor of the activation value as including some empirical heuristics of [Tsuda *et al.* 89a]. There may, however, be more effective heuristics with more factors or with a non-linear formula[Hasida 91].

5.2 Example of cu-PrologIII

Figure 1 is an example of disjunctive feature unification in [Kasper 87].

Figure 2 is an example of the JPSG parser in cu-PrologIII. For ambiguous sentences, the parser returns the corresponding feature structure with constraints.

6 Concluding Remarks

This paper outlined cu-Prolog, then covered the disjunctive feature structure and parsing JPSG.

We would like to stress that every feature mentioned in this paper was equally processed in the same framework as a constraint transformation. In comparison with many conventional approaches, our approaches, including Hasida's DP (Dependency Propagation) [Hasida 91], are far more general and flexible frameworks for constraint-based natural language processing.

Acknowledgment

The author thanks Hidetosi Sirai of Chukyo University for his cooperation in implementing cu-Prolog. Thanks are also due to Kazumasa Yokota, Hideki Yasukawa, and Kôiti Hasida and other members of ICOT for their comments.

References

[Carpenter *et al.* 91] Bob Carpenter, Carl Pollard, and Alex Franz. The Specification and Implementation of Constraint-Based Unification Grammar. In *Proc. of Second International Workshop on Parsing Technologies*, pages 143–153. Sigparse ACL, February 1991.

[Chomsky 81] Norm Chomsky. *Lectures on Government and Binding.* Foris, Dordrecht, 1981.

[Eisele and Dörre 88] Andreas Eisele and Jochen Dörre. Unification of Disjunctive Feature Descriptions. In *Proc. of 26th Annual Meeting of ACL*, pages 286–294, June 1988.

[Gunji 86] Takao Gunji. *Japanese Phrase Structure Grammar.* Reidel, Dordrecht, 1986.

[Hasida 91] Kôiti Hasida. Common Heuristics for Parsing, Generation, and Whatever. In *Workshop on Reversible Grammar in Natural Language Processing*, Berkeley, 1991.

[Hasida and Sirai 86] Kôiti Hasida and Hidetosi Sirai. Conditioned Unification. *Computer Software*, 3(4):28–38, 1986. (in Japanese).

[Jaffar and Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proc. of 14th ACM POPL Conference*, pages 111–119, Munich, 1987.

[Kasper 87] Robert T. Kasper. A Unification Method for Disjunctive Feature Descriptions. In *Proc. of 25th Annual Meeting of ACL*, pages 235–242. July 1987.

[Kasper and Rounds 86] Robert T. Kasper and William C. Rounds. A Logical Semantics for Feature Structure. In *Proc. of 24th ACL Annual Meeting*, pages 257–266, 1986.

[Kay 85] Martin Kay. Parsing in Functional Unification Grammar. In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing*, chapter 7, pages 251–278. Cambridge University Press, 1985.

[Mukai 88] Kuniaki Mukai. Partially Specified Term in Logic Programming for Linguistic Analysis. In *Proc. of the International Conference of Fifth Generation Computer Systems*, pages 479–488. ICOT, OHMSHA and Springer-Verlag, 1988.

[Pollard and Sag 87] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol.1 Fundamentals.* CSLI Lecture Notes Series No.13. Stanford:CSLI. 1987.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-Based Approach to Grammar.* CSLI Lecture Notes Series No.4. Stanford:CSLI. 1986.

[Sirai 91] Hidetosi Sirai. A Guide to MacCUP. unpublished. 1991. (available by anonyous ftp from csli.stanford.edu (pub/MacCup)).

[Tamaki and Sato 83] Hisao Tamaki and Taisuke Sato. Unfold/Fold Transformation of Logic Programs. In *Proc. of Second International Conference on Logic Programming*, pages 127–137, 1983.

[Tsuda 91] Hiroshi Tsuda. Disjunctive Feature Structure in cu-Prolog. In *8th Conf. Proc. of Japan Society of Software Science and Technology.* pages 505–508. 1991. (in Japanese)

[Tsuda and Hasida 90] Hiroshi Tsuda and Kôiti Hasida. Parsing as Constraint Transformation — an Extension of cu-Prolog. In *Proc. of the Seoul International Conference on Natural Language Processing*, pages 325–331. 1990.

[Tsuda *et al.* 89a] Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. cu-Prolog and its Application to a JPSG Parser. In K.Furukawa, H.Tanaka, and T.Fujisaki, editors, *Logic Programming '89*, pages 134–143. Springer-Verlag LNAI-485, 1989.

[Tsuda *et al.* 89b] Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. JPSG Parser on Constraint Logic Programming. In *Proc. of 4th ACL European Chapter*, pages 95–102. 1989.

[Tsuda *et al.* 92] Hiroshi Tsuda, Kôiti Hasida, and Hidetosi Sirai. cu-PrologIII System. ICOT Technical Memorandum. ICOT TM-1160. 1992.

```
cc1({voice/passive,trans/trans,subj/X,goal/X}). % definition of the unconditional conjuncts
cc1({voice/active, subj/X,actor/X}).
cc2({trans/intrans, actor/{person/third}}).
cc2({trans/trans, goal/{person/third}}).
cc3({numb/sing, subj/{numb/sing}}).
cc3({numb/pl, subj/{numb/pl}}).

% disjunctive feature unification (user input)
@ U={rank/clause, subj/{case/nom}}, cc1(U),cc2(U),cc3(U), U={subj/{lex/you,person/second,numb/pl}}.

% answer: equivalent constraint
solution = c0(U_0, {subj/{case/nom}, rank/clause}, {subj/{person/second, numb/pl, lex/you}})

% definitions of a new predicate (c0)
c0(_p1, _p1, _p1) :- cc2(_p1), cc1(_p1);
  _p1={subj/{person/second, numb/pl, case/nom, lex/you}, numb/pl, rank/clause}.

CPU time = 0.150 sec (Constraints Handling = 0.000 sec)

>:-c0(X,_,_).        % solve the new constraint
success.             % X is the final answer of the unification.
 X = {voice/active, trans/trans, subj/{person/second, numb/pl, case/nom, lex/you},
 goal/{person/third}, actor/{person/second, numb/pl, case/nom, lex/you}, numb/pl, rank/clause};
```

Lines beginning with "@" are user inputs. To this input, cu-Prolog returns equivalent modular constraint and definition clauses of newly defined predicates.

Figure 1: Disjunctive feature unification

```
_:-p([ken,ga,ai,suru]).      % user input of ''Ken ga ai-suru.''

    %%% parse tree
{sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v}, sc/V1_2024,
 refl/[], slash/V3_2026, psl/[], ajn/[], ajc/[]}---[suff_p]
 |
 |--{sem/[love,V7_2030,V6_2029], core/{pos/v}, sc/V0_2023, refl/[],
   slash/V2_2025, psl/[], ajn/[], ajc/[]}---[subcat_p]
 |  |
 |  |--{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
      psl/[], ajn/[], ajc/[]}---[adjacent_p]
 |  |  |
 |  |  |--{sem/ken, core/{form/n, pos/n}, sc/[], refl/[], slash/[],
        psl/[], ajn/[], ajc/[]}---[ken]
 |  |  |
 |  |  |__{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],psl/[], ajn/[],
       ajc/[{sem/ken, core/{pos/n}, sc/[], refl/Refl1AC_70}]}---[ga]
 |  |
 |  |__{sem/[love,V7_2030,V6_2029], core/{form/vs2, pos/v}}---[ai]
 |
 |__{sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v}, sc/[], refl/[],
   slash/[], psl/[], ajn/[], ajc/[{sem/[love,V7_2030,V6_2029],
   core/{form/vs2, pos/v}, sc/[], refl/Refl1AC_1493}]}---[suru]

category= {sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v},
    sc/V1_2024, refl/[], slash/V3_2026, psl/[], ajn/[], ajc/[]} %category

constraint= c40(V0_2023, V1_2024, V2_2025, V3_2026, V4_2027, V5_2028,
     {sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[], psl/[],
     ajn/[], ajc/[]}, V6_2029, {sem/V6_2029, core/{form/wo, pos/p}}, V7_2030,
     {sem/V7_2030, core/{form/ga, pos/p}}),
  syu_ren(Form_1381) %constraint about the category
true.
CPU time = 2.217 sec (Constraints Handling = 1.950 sec)

_:-c40(V1, _, _, V3, _, _,_, V6,_ , V7,_).      %solve constraint
 V1 = []  V3 = [{sem/V0_4}]  V6 = V0_4  V7 = ken;     % solution 1
 V1 = [{sem/V0_4, core/{form/wo, pos/p}}]  V3 = []  V6 = V0_4  V7 = ken;  % solution 2
no.
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)
```

The parsing of "Ken ga ai-suru" that has two meanings: "Ken loves (someone)" or "(someone) whom Ken loves." The parser draws a corresponding parse tree and returns the category of the top node with constraints. In this example, the ambiguity of the sentence is shown in the two solutions of the constraint c40.

Figure 2: JPSG parser: disambiguation

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Model Generation Theorem Provers on a Parallel Inference Machine

Masayuki Fujita Ryuzo Hasegawa
Miyuki Koshimura* Hiroshi Fujita†

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{mfujita, hasegawa, koshi}@icot.or.jp fujita@sys.crl.melco.co.jp

Abstract

This paper describes the results of the research and development on parallel theorem provers being conducted at ICOT. We have implemented a model-generation based parallel theorem prover called MGTP in KL1 on a distributed memory multi-processor, Multi-PSI, and on a parallel inference machine with the same architecture, PIM/m. Currently, we have two versions of MGTP: one is MGTP/G, which is used for dealing with ground models, and the other is MGTP/N, used for dealing with non-ground models. While conducting research and development on the MGTP provers, we have developed several techniques to improve the efficiency of forward reasoning theorem provers. These include model generation and hyper-resolution theorem provers. First, we developed KL1 compilation techniques to translate the given clauses to KL1 clauses, thereby achieving good efficiency. To avoid redundancy in conjunctive matching, we developed RAMS, MERC, and Δ-M methods. To reduce the amount of computation and space required for obtaining proofs, we proposed the idea of *Lazy Model Generation*. Lazy model generation is a new method that avoids the generation of unnecessary atoms that are irrelevant to obtaining proofs, and provides flexible control for the efficient use of resources in a parallel environment. For MGTP/G, we exploited OR parallelism with a simple allocation scheme, thereby achieving good performance on the Multi-PSI. For MGTP/N, we exploited AND parallelism, which is rather harder to obtain than OR parallelism. With the lazy model generation method, we have achieved a more than one-hundred-fold speedup on a PIM/m consisting of 128 PEs.

*Present address: Toshiba Information Systems
2-1 Nissin-cho, Kawasaki-ku, Kawasaki, Kanagawa 210, Japan
†Present address: Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-honmachi, Amagasaki, Hyogo 661, Japan

1 Introduction

The research on parallel theorem proving systems has been conducted under the Fifth Research Laboratory at ICOT as a part of research and development on the problem-solving programming module. This research aims at the realization of highly parallel advanced inference mechanisms that are indispensable in building intelligent knowledge information systems.

The immediate goal of this research project is to develop a parallel automated reasoning system on the parallel inference machine, PIM, based on KL1 and PIMOS technology [Chikayama *et. al.* 88]. We aim at applying this system to various fields such as intelligent database systems, natural language processing, and automated programming.

The motive for the research is twofold.

From the viewpoint of logic programming, we try to further extend logic programming techniques that provide the foundation for the Fifth Generation Computer System. The research will help those aiming at extending languages and/or systems from Horn clause logic to full first-order logic. In addition, theorem proving is one of the most important applications that could effectively be built upon the logic programming systems. In particular, it is a good application for evaluating the abilities of KL1 and PIM.

From the viewpoint of automated reasoning, on the other hand, it seems that the logic programming community is ready to deal with more classical and difficult problems[Wos *et. al.* 84][Wos 88] that remain unsolved or have been abandoned. We might achieve a breakthrough in the automated reasoning field if we apply logic programming technology to theorem proving. In addition, this trial would also cause feedback for logic programming technology.

Recent developments in logic programming languages and machines have shed light upon the problem of how to implement these classical but powerful methods efficiently. For instance, Stickel developed a model-

elimination[Loveland 78] based theorem prover called PTTP[Stickel 88]. PTTP is able to deal with any first-order formula in Horn clause form (augmented by contrapositives) without loss of completeness or soundness. It works by employing unification with occurrence check, the model elimination reduction rule, and iterative deepening depth-first search. A parallel version of PTTP, called PARTHENON[Bose *et. al.* 89], has been implemented by Clarke et al. on a shared memory multiprocessor. Schumann et al. built a connection-method[Bibel 86] based theorem-proving system, SETHEO[Schumann 89], in which a method identical to model elimination is used as a main proof mechanism. Manthey and Bry presented a tableaux-like theorem prover, SATCHMO[Manthey and Bry 88], which is a very short and simple program in Prolog.

As a first step for developing KL1-technology theorem provers, we adopted the model generation method on which SATCHMO is based. Our reasons were as follows:

(1) A useful feature of SATCHMO is that full unification is not necessary, and that matching suffices when dealing with range-restricted problems. This makes it very convenient for us to implement provers in KL1 since KL1, as a committed choice language, provides us with very fast one-way unification.

(2) It is easier to incorporate mechanisms for lemmatization, subsumption tests, and other deletion strategies that are indispensable in solving difficult problems such as condensed detachment problems [Wos 88][Overbeek 90][McCune and Wos 91].

In implementing model generation based provers, it is important to avoid redundancy in the *conjunctive matching* of clauses against atoms in model candidates. For this, we proposed the RAMS [Fujita and Hasegawa 91] and MERC [Hasegawa 91a] methods.

A more important issue with regard to the efficiency of model generation based provers is how to reduce the total amount of computation and memory required for proof processes. This problem becomes more critical if we try to solve harder problems that require deeper inferences (longer proofs) such as Lukasiewicz problems. To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes and that generation should be performed only when testing requires it. We proposed the *Lazy Model Generation* method in which the idea of demand-driven computation or 'generate-only-at-test' is implemented. Lazy model generation is a new method that avoids the generation of unnecessary atoms that are irrelevant to obtaining proofs, and provides flexible control for the efficient use of resources in a parallel environment.

We have implemented two types of model generation prover: one is used for ground models (MGTP/G) and the other is used for non-ground models (MGTP/N).

In implementing MGTP/G, we developed a compiling technique to translate the given clauses into KL1 clauses by using advantage (1) listed above. This makes MGTP/G very simple and efficient. MGTP/G can prove non-Horn problems very efficiently on a distributed memory multi-processor, the Multi-PSI, by exploiting OR parallelism.

MGTP/N, on the other hand, aims at proving difficult Horn problems by exploiting AND parallelism. For MGTP/N, we developed new parallel algorithms based on lazy model generation method. They run with optimal load balancing on a distributed memory architecture, and require a minimal amount of computation and memory to obtain proofs.

In the next section, we explain the model generation method on which our MGTP provers are based. In Section 3, we discuss the problem of meta-programming in KL1, and outline the characteristics of MGTP/G and MGTP/N. In Section 4, we describe the essence of the main techniques developed for improving the efficiency of model generation theorem provers. In Section 5, we present OR parallelization and AND parallelization methods developed for MGTP/G and MGTP/N. Section 6 provides a conclusion.

2 Model Generation Theorem Prover

Throughout this paper, a clause is represented in an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; the consequent is a disjunction of $C_1, C_2, \ldots, C_m$. A clause is said to be *positive* if its antecedent is $true (n = 0)$, and *negative* if its consequent is $false (m = 0)$. A clause is also said to be *tester* if its consequent is $false (m = 0)$, otherwise it is called *generator*.

The model generation method incorporates the following two rules:

- Model extension rule: If there is a generator clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model candidate M and $C\sigma$ is not satisfied in M, then extend M by adding $C\sigma$ into M.

- Model rejection rule: If a tester clause has an antecedent $A\sigma$ that is satisfied in a model candidate M, then reject M.

We call the process of obtaining $A\sigma$ a *conjunctive matching* of the antecedent literals against elements in a model. Note that the antecedent (*true*) of a positive clause is satisfied by any model.

The task of model generation is to try to construct a model for a given set of clauses, starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found. The method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to see that no model exists for the clause set.

For example, consider the following set of clauses, S1[Manthey and Bry 88]:

$$
\begin{aligned}
C1: &\quad p(X), s(X) \rightarrow false. \\
C2: &\quad q(X), s(Y) \rightarrow false. \\
C3: &\quad q(X) \rightarrow s(f(X)). \\
C4: &\quad r(X) \rightarrow s(X). \\
C5: &\quad p(X) \rightarrow q(X); r(X). \\
C6: &\quad true \rightarrow p(a); q(b).
\end{aligned}
$$

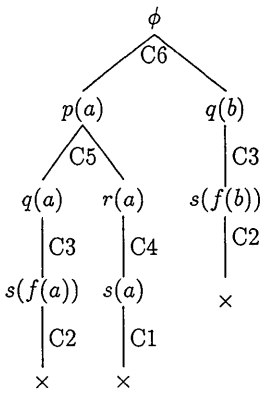


Figure 1: A proof tree for S1

A proof tree for the S1 problem is depicted in Fig. 1. We start with an empty model, $M_0 = \phi$. M_0 is first expanded into two cases, $M_1 = \{p(a)\}$ and $M_2 = \{q(b)\}$, by applying the model extension rule to $C6$. Then M_1 is expanded by $C5$ into two cases: $M_3 = \{p(a), q(a)\}$ and $M_4 = \{p(a), r(a)\}$. M_3 is further extended by $C3$ to $M_5 = \{p(a), q(a), s(f(a))\}$. Now with M_5 the model rejection rule is applicable to $C2$, thus M_5 is rejected and marked as closed. On the other hand, M_4 is extended by $C4$ to $M_6 = \{p(a), r(a), s(a)\}$ which is rejected by $C1$. In a similar way, the remaining model candidate M_2 is extended by $C3$ to $M_7 = \{q(b), s(f(b))\}$, which is rejected by $C2$. Now that there is no way to construct any model candidate, we can conclude that the clause set S1 is unsatisfiable.

The model generation method, as its name suggests, is closely related to the model elimination method. However, the model generation method is a restricted version of the model elimination method in the sense that the polarity of literals in a clause of implicational form is fixed to either positive or negative in the model generation method, whereas it is allowed to be both positive and negative in the model elimination method. Moreover, from the procedural point of view, model generation is restricted to proceeding bottom-up (as in forward-reasoning) starting at positive clauses (or facts). These restrictions, however, do not hurt the refutation completeness of the method.

Model generation can also be viewed as *unit hyper-resolution*. Our calculus, however, is much closer to tableaux calculus in the sense that it explores a tree, or a tableau, in the course of finding a proof. Indeed, a branch in a proof tree obtained by the tableaux method corresponds exactly to a model candidate.

3 Two Versions of MGTP

3.1 Meta-programming in KL1

Prolog-Technology Theorem Provers such as PTTP and SATCHMO utilize the fact that Horn clause problems can be solved very efficiently. In these systems, the theorem being proven is represented by Prolog clauses, and most deductions are performed as normal Prolog execution. However, that approach cannot be taken in KL1 because a KL1 clause is not just a Horn clause; it has extra-logical constructs such as a guard and a commit operator.

We should, therefore, treat the clause set as data rather than as a KL1 program. In this case, the inevitable problem is how to represent variables appearing in a given clause set. Two approaches can be considered for this problem:

(1) representing object-level variables with KL1 ground terms, or

(2) representing object-level variables with KL1 variables.

The first approach might be the right path in meta-programming, where object- and meta-levels are strictly separated, thereby providing clear semantics. However, it forces us to write routines for unification, substitution, renaming, and all the other intricate operations on variables and environments. These routines would become extremely large and complex compared to the main program, and would make the overhead bigger.

In the second approach, most operations on variables and environments can be performed beside the underlying system, rather than as routines running on top of it. This means that a meta-programmer does not have to write tedious routines, and gains high efficiency.

Also, a programmer can use the Prolog var predicate to write routines such as occurrence checks in order to

make built-in unification sound, if such routines are necessary. This approach makes the program much more simple and efficient, even though it makes the distinction between object- and meta-levels ambiguous.

In KL1, however, the second approach is not always possible. This is because the semantics of KL1 never allow us to use a predicate like var. In addition, KL1 built-in unification is not the same as its Prolog counterpart in that unification in the guard part of a KL1 clause can only be one-way, and a unification failure in the body part is regarded as a program error or exception that cannot be backtracked.

3.2 Characteristics of MGTP/G and MGTP/N

Taking the above discussions into consideration, we decided to develop both the MGTP/G and MGTP/N provers so that we can use effectively them according to the problem domains dealt with.

The ground version, MGTP/G, aims to support finite problem domains, which include most problems in various fields, such as database processing and natural language processing.

For ground model cases, the model generation method makes it possible to use just matching, rather than full unification, if the problem clauses satisfy the *range-restrictedness* condition [1] [Manthey and Bry 88]. This suggests that it is sufficient to use KL1's head unification. Thus we can take the KL1 variable approach for representing object-level variables, thereby achieving good performance.

The key points of KL1 programming techniques developed for MGTP/G are as follows: (Details are described in the next section.)

- First, we translate a given set of clauses into a corresponding set of KL1 clauses. This translation is quite simple.
- Second, we perform conjunctive matching of a literal in a clause against a model element by using KL1 head unification.
- Third, at the head unification, we can automatically obtain fresh variables for a different instance of the literal used.

The non-ground version, MGTP/N, supports infinite problem domains. Typical examples are mathematical theorems, such as group theory and implicational calculus.

[1] A clause is said to be range-restricted if every variable in the clause has at least one occurrence in its antecedent. For example, in the S1 problem, all the clauses, $C1$-$C6$, are range-restricted since no variable appears in clause $C6$; the variable X in clauses $C1, C3, C4$ and $C5$ has an occurrence in their antecedents; and variables X and Y in $C2$ have their occurrences in its antecedent.

```
c(1,p(X),[],    R):-true|R=cont.
c(1,s(X),[p(X)],R):-true|R=false.
c(2,q(X),[],    R):-true|R=cont.
c(2,s(Y),[q(X)],R):-true|R=false.
c(3,q(X),[],    R):-true|R=[s(f(X))].
c(4,r(X),[],    R):-true|R=[s(X)].
c(5,p(X),[],    R):-true|R=[q(X),r(X)].
c(6,true,[],    R):-true|R=[p(a),q(b)].
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 2: S1 problem transformed to KL1 clauses

For non-ground model cases, where full unification with occurrence check is required, we are forced to follow the KL1 ground terms approach. However, we do not necessarily have to maintain variable-binding pairs as processes in KL1. We can maintain them by using the vector facility supported by KL1, as is often used in ordinary language processing systems. Experimental results show that vector implementation is several hundred times faster than process implementation.

In this case, however, we cannot use the programming techniques developed for MGTP/G. Instead, we have to use a conventional technique, that is, interpreting a given set of clauses instead of compiling it into KL1 clauses.

To ease the programmer's burden, we developed *Meta-Library*[Koshimura *et. al.* 90]. This is a collection of KL1 programs to support meta-programming in KL1. The meta-library includes facilities such as full unification with occurrence check, variable management routines, and term memory[Stickel 89][Hasegawa 91c].

4 Technologies Developed for Efficiency

4.1 KL1 Compiling Method

This section presents the compiling techniques developed for MGTP/G to translate given clauses to KL1 clauses. It also shows a simple MGTP/G interpreter obtained by using the techniques[Fuchi 90][Fujita and Hasegawa 90][Hasegawa *et. al.* 90a].

4.1.1 Transforming problem clauses to KL1 clauses

Our MGTP/G prover program consists of two parts: an interpreter written in KL1, and a set of KL1 clauses representing a set of clauses for the given problem. During conjunctive matching, an antecedent literal expressed in the head of a KL1 clause is matched against a model element chosen from a model candidate which is retained in the interpreter.

Although conjunctive matching can be implemented simply in KL1, we need a programming trick for support-

ing variables shared among literals in a problem clause. The trick is to propagate the binding for a shared variable from one literal to another.

To understand this, consider the previous example, S1. The original clause set is transformed into a set of KL1 clauses, as shown in Figure 2. In `c(N,P,S,R)`, `N` indicates clause number; `P` is an antecedent literal to be matched against an element taken from a model candidate; `S` is a pattern for receiving from the interpreter a stack of literal instances appearing to the left of `P`, which have already matched model elements; and `R` is the result returned to the interpreter when the match succeeds.

Notice that original clause $C1$ $(p(X), s(X) \rightarrow false.)$ is translated to the first two KL1 clauses. The conjunctive matching for $C1$ proceeds as follows. First, the interpreter picks up a model element, E_1, from a model candidate, and tries to match the first literal $p(X)$ in C1 against E_1 by initiating a goal, $c(1, E_1, [], R_1)$. If the matching fails, then the result $R_1 = fail$ is returned by the last KL1 clause. If the matching succeeds, then the result $R_1 = cont$ is returned by the first KL1 clause and the interpreter proceeds to the next literal $s(X)$ in C1, picking up another model element, E_2, from the model candidate and initiating a goal, $c(1, E_2, [E_1], R_2)$. Since the literal instance in the third argument, $[E_1]$, is ground, the variable X in $[p(X)]$ in the head of the second KL1 clause gets instantiated to a ground term. At the same time, the term $s(X)$ in that head is also instantiated due to the shared variable X. Under this instantiation, $s(X)$ is checked to see whether it matches E_2, and if the matching succeeds then the result, $R_2 = false$, is returned.

4.1.2 A simple MGTP/G interpreter

With the problem clauses are transformed to KL1 clauses as above, a simple interpreter is developed as shown in Figure 3[2].

The interpreter, given a list of numbers identifying problem clauses and a model candidate, checks whether the clauses are satisfiable or not. The top-level predicate, `clauses/5`, dispatches a task, `ante/7`, to check whether each clause is satisfied or not in the current model. If all the clauses are satisfied in the current model, the result, `sat`, is returned by `sat/4` combining the results from the `ante` processes.

For each clause in the given clauses, conjunctive matching is performed between the elements in the model candidate and the literals in the antecedent of the clause with `ante/7` and `ante1/9` processes. The conjunctive matching for the antecedent literals proceeds from left to right, by calling `c/4` one by one. An `ante` process retains a stack, S, of literal instances. If the match succeeds at a literal, L_i, with a model element, P, then P is pushed onto the stack S, and the task proceeds to matching the next literal, L_{i+1}, together with the stack, $[P|S]$.

According to the result of `c/4`: `fail`, `cont`, `false` or `list(F)`, an `ante1/9` process determines what to do next. If the result is `cont`, for example, `ante1` will fork multiple `ante` processes to try to make every possible combination of elements out of the current model for the conjunctive matching.

If the conjunctive matching for all the antecedent literals of a clause succeeds, a `cnsq/6` process is called to check the satisfiability of the consequent of the clause. `cnsq1/8` checks whether a literal in the consequent is a member of the current model. If no literal in the consequent is a member of the current model, the current model cannot satisfy the clause. In this case, the model will be extended with each disjunct literal in the consequent of the clause by calling an `extend/5` process.

After extending the current model, a `clauses/5` process is called for each extension of the model, and the results are combined by `unsat/4`. When a `clauses` process for some of the extended models returns `sat` as the result, it means that a model is found and the clause set is known to be satisfiable. If every extension of the model leads to `unsat`, the current model is not a part of any model for the given set of clauses.

Thus, if the top-level `clauses/5` process returns `sat` as the result, then the given clause set has a model and is satisfiable, and if it returns `unsat`, then the given clause set has no model and is unsatisfiable.

4.2 Avoiding Redundant Conjunctive Matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching.

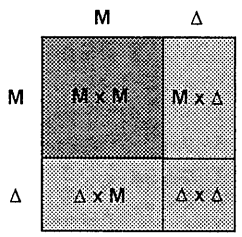
Let us consider a clause, C, having two antecedent literals. To perform conjunctive matching for the clause, we need to pick a pair of atoms out of the current model candidate, M. Imagine that, as a result of a satisfiability check of the clause, we are to extend the model candidate with Δ, which is an atom in the consequent of the clause, C, but not in M. Then, in the conjunctive matching for the clause, C, in the next phase, we need to pick a pair of atoms from $M \cup \Delta$. The number of pairs amounts to:

$$(M \cup \Delta)^2 = M \times M \cup M \times \Delta \cup \Delta \times M \cup \Delta \times \Delta.$$

[2] In the program, '`alternatively`' is a KL1 compiler directive which gives a preference among clauses to evaluate their guards in such a way that clauses above `alternatively` are evaluated before those below it. The preference, however, may not be strictly obeyed. This depends on implementation.

```
clauses(_,_,_,_,quit):-true|true.
alternatively.
clauses([J|Cs],C,M,A,B):-true|
    ante(J,[true|M],[],C,M,A1,B),
    sat(A1,A2,A,B), clauses(Cs,C,M,A2,B).
clauses([],_,_,A,_):-true|A=sat.

ante(_,_,_,_,_,_,quit):-true|true.
alternatively.
ante(J,[P|M2],S,C,M,A,B):-true|
    mgtp:c(J,P,S,R),
    ante1(J,R,P,S,M2,C,M,A,B).
ante(_,[],_,_,_,A,_):-true|A=sat.

ante1(J,fail,_,S,M2,C,M,A,B):-true|
    ante(J,M2,S,C,M,A,B).
ante1(J,cont,P,S,M2,C,M,A,B):-true|
    ante(J,M,[P|S],C,M,A1,B),
    sat(A1,A2,A,B), ante(J,M2,S,C,M,A2,B).
ante1(_,false,_,_,_,_,M,A,B):-true|
    A=unsat,B=quit.
ante1(J,F,_,S,M2,C,M,A,B):-list(F)|
    cnsq(F,F,C,M,A1,B),
    sat(A1,A2,A,B), ante(J,M2,S,C,M,A2,B).

cnsq(_,_,_,_,_,_,quit):-true|true.
alternatively.
cnsq([D1|Ds],F,C,M,A,B):-true|
    cnsq1(D1,M,Ds,F,C,M,A,B).
cnsq([],F,C,M,A,_):-true|
    extend(F,M,C,A,_).

cnsq1(D,[D|_],_,_,_,_,_,A,_):-true|A=sat.
cnsq1(_,[],Ds,F,C,M,A,B):-true|
    cnsq(Ds,F,C,M,A,B).
otherwise.
cnsq1(D,[_|M2],Ds,F,C,M,A,B):-true|
    cnsq1(D,M2,Ds,F,C,M,A,B).

extend(_,_,_,_,_,quit):-true|true.
alternatively.
extend([D|Ds],M,C,A,B):-true|
    clauses(C,C,[D|M],A1,_),
    unsat(A1,A2,A,B), extend(Ds,M,C,A2,B).
extend([],_,_,A,_):-true|A=unsat.

sat(sat,sat,A,_):-true|A=sat.
sat(unsat,_,A,B):-true|A=unsat,B=quit.
sat(_,unsat,A,B):-true|A=unsat,B=quit.

unsat(unsat,unsat,A,_):-true|A=unsat.
unsat(sat,_,A,B):-true|A=sat,B=quit.
unsat(_,sat,A,B):-true|A=sat,B=quit.
```

Figure 3: A simple MGTP/G interpreter



It should be noted here that $M \times M$ pairs were already considered in the previous phase of conjunctive matching. If they were chosen in this phase, the result would contribute nothing since the model candidate need not be extended with the same Δ. Hence, redundant consideration on $M \times M$ pairs should be avoided at this time. Instead, we have to choose only the pairs which contain at least one Δ. This discussion can be generalized for cases in which we have more than two antecedent literals, any number of clauses, and any number of model candidates.

We have taken two approaches to avoid the above redundancy. One approach uses a stack to keep the intermediate results obtained by matching a literal against an element out of the model candidate. The other approach recomputes the intermediate matching results without keeping them.

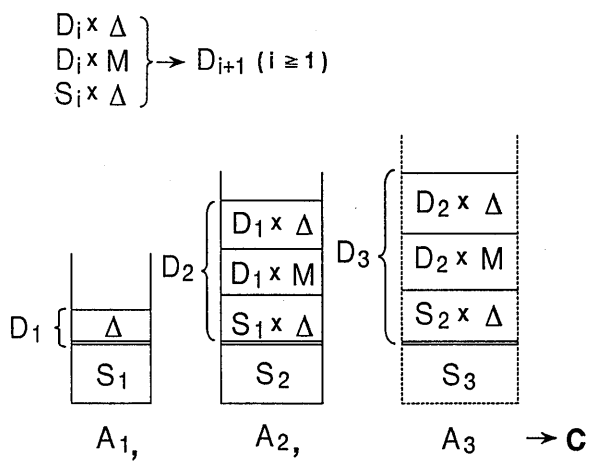


Figure 4: RAMS method

4.2.1 RAMS Method

The RAMS (ramified-stack) method [Hasegawa *et. al.* 90a][Hasegawa *et. al.* 90b][Fujita and Hasegawa 91] retains in a stack an instance which is a result of matching a literal against a model element. The use of this method

for a Horn clause case is illustrated in Figure 4, where M is a model candidate and Δ is an atom picked from a model-extending candidate.

- A stack called *a literal instance stack* (LIS), is assigned to each antecedent literal, A_i, in a clause for storing literal instances. Note that LIS for the last literal expressed in dashed boxes needs not actually be allocated.
- LIS is divided into two parts: D_i and S_i where $D_i (i \geq 1)$ is a set of literal instances generated at the current stage triggered by Δ; and S_i is those created in previous stages.
- A task, being performed at each literal, A_i, computes the following:

$$D_1 := \Delta;$$

$$D_{i+1} := D_i \times \Delta \cup D_i \times M \cup S_i \times \Delta \ (i \geq 1)$$

where $A \times B$ denotes a set of pairs of an instance taken from A and B. The above tasks are performed from left to right.

For non-Horn clause cases, each LIS branches to make a tree-structured stack when case splitting occurs. The name 'RAMS' comes from this. The idea is as follows:

- A model is represented by a branch of a ramified stack, and the model is extended only at the top of the current stack.
- After applying the model extension rule to a non-Horn clause, the current model may be extended to multiple descendant models.
- Every descendant model that is extended from a parent model can share its ancestors with other sibling models just by pointing to the top of the stack corresponding to the parent.
- Each descendant model can extend the stack for itself, independent of other sibling models.

The ramified-stack method not only avoids redundancy in conjunctive matching but also enables us to share a common model. However, it has one drawback: it tends to require a lot of memory to retain intermediate literal instances.

4.2.2 MERC Method

The MERC (Multi-Entry Repeated Combination) method [Hasegawa 91a] tries to solve the above problem using the RAMS method. This method does not need a memory to retain intermediate results obtained in the conjunctive matching. Instead, it needs to prepare $2^n - 1$ clauses for the given clause having n literals as its antecedent.

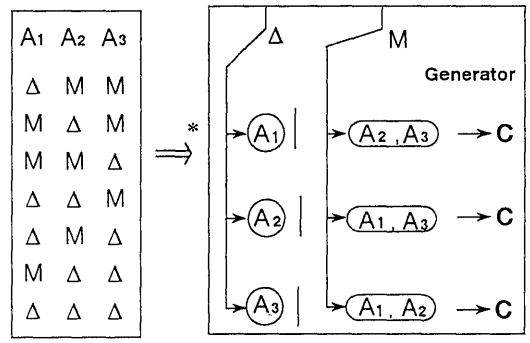


Figure 5: MERC method

An outline of the MERC method is shown in Figure 5. For a clause having three antecedent literals, $A_1, A_2, A_3 \rightarrow C$, we prepare seven clauses. Each of these clauses corresponds to a repeated combination of Δ and M, and performs conjunctive matching using the combination pattern. For example, a clause corresponding to a combination pattern $[M, \Delta, M]$ first matches literal A_2 against Δ. If the match succeeds, it proceeds to match the remaining literals, A_1 and A_3, against an element picked from M. Note that each combination pattern includes at least one Δ, and that the $[M, M, M]$ pattern is excluded.

For ground model cases, optimization can be used to reduce the number of clauses by testing the unifiability of antecedent literals. For example, if any antecedent literal in the given clause is not unifiable with the other antecedent literal in that clause, it is sufficient to consider the following three combination patterns: $[\Delta, M, M]$,$[M, \Delta, M]$ and $[M, M, \Delta]$. The right-hand side in Figure 5 shows the clauses obtained after making the unifiability test.

4.2.3 Δ-M Method

The problem with the MERC method is that the number of prepared clauses increases exponentially as the number of antecedent literals increases. In actual implementation, we adopted a modified version of the MERC method, which we call the Δ-M method. In place of multiple entry clauses, the Δ-M method prepares a template like:

$$\{[\Delta, \Delta], [\Delta, M], [M, \Delta]\}$$

for clauses with two antecedent literals, and

$$\{[\Delta, \Delta, \Delta], [\Delta, \Delta, M], [\Delta, M, \Delta], [M, \Delta, \Delta],$$

$$[\Delta, M, M], [M, \Delta, M], [M, M, \Delta]\}$$

for clauses with three antecedent literals, and so forth. According to this pattern, we enumerate all possible combinations of atoms for matching the antecedent literals of given clauses.

There are some trade-offs between the RAMS method and the MERC and Δ-M methods. In the RAMS method, every successful result of matching a literal A_i against model elements is memorized so that the same literal is not rematched against the same model element. On the other hand, both the MERC and Δ-M methods do not need to memorized information on partial matching. However, they still contain a redundant computation. For instance, in the computation for $[M, \Delta, \Delta]$ and $[M, \Delta, M]$ patterns, the common subpattern $[M, \Delta]$, will be recomputed. The RAMS method can eliminate this sort of redundancy.

4.3 Lazy Model Generation

Model-generation based provers must perform the following three operations.

- create new model elements by applying the model extension rule to the given clauses using a set of model-extending atoms Δ and a model candidate set M (model extension).
- make a subsumption test for a created atom to check if it is subsumed by the set of atoms already being created, usually by the current model candidate.
- make a false check to see if the unsubsumed model element derives false by applying the model extension rule to the tester clauses (rejection test).

The problem with the model generation method is the huge growth in the number of generated atoms and in the computational cost in time and space, which is incurred by the generation processes.

To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes, and that generation should be performed only when testing requires it.

For this we proposed a lazy model generation algorithm [Hasegawa 91b][Hasegawa 91d][Hasegawa *et. al.* 92a][Hasegawa *et. al.* 92b] that can reduce the amount of computation and space necessary for obtaining proofs.

This section presents several algorithms, including the lazy algorithm, for the model generation method, and compares them in terms of time and space. To simplify the presentation, we assume that the problem is given only in Horn clauses. However, the principle behind these algorithms can be applicable to non-Horn clauses as well.

4.3.1 Basic Algorithm

The basic algorithm shown in Figure 6 performs model generation with a search strategy in a breadth-first fashion. This is essentially the same algorithm as the hyper-resolution algorithm taken by OTTER [McCune 90] [3].

```
M := φ;
D := {A | (true → A) ∈ a set of given clauses};
while D ≠ φ do begin
      D := D − Δ;
      if CJM_Tester(Δ, M) ∋ false
            then return(success);
      new := CJM_Generator(Δ, M);
      M := M ∪ Δ;
      new' := subsumption(new, M ∪ D);
      D := D ∪ new';
end return(fail)
```

Figure 6: Basic algorithm

In the algorithm, M represents model candidate , D represents the model-extending candidate (a set of model-extending atoms which are generated as a result of the application of the model extension rule and are going to be added to M), and Δ represents a subset of D. Initially, M is set to an empty set, and D is a set of positive (unit) clauses of the given problem.

In each cycle of the algorithm,

1) Δ is selected from D,
2) a rejection test (conjunctive matching for the tester clauses) is performed on Δ and M,
3) if the test succeeds then the algorithm terminates,
4) if the test fails then model extension (conjunctive matching on the generator clauses) is performed on Δ and M, and
5) a subsumption test is performed on *new* against $M \cup D$.

If D is empty at the beginning of a cycle, then the algorithm terminates as the refutation fails (In other words, a model is found for the given set of clauses).

The conjunctive matching and subsumption test is represented by the following functions on sets of atoms.

$$CJM_{Cs}(\Delta, M) = \{\sigma C \mid \sigma A_1, \ldots, \sigma A_n \rightarrow \sigma C \wedge A_1, \ldots, A_n \rightarrow C \in Cs \wedge \sigma A_i = \sigma B (B \in M \cup \Delta)(1 \leq \forall i \leq n) \wedge \exists i (1 \leq i \leq n) \sigma A_i = \sigma B (B \in \Delta)\}$$

$$subsumption(\Delta, M) = \{C \in \Delta \mid \forall B \in M (B \text{ dosen't subsume } C)\}$$

[3] OTTER is a slightly optimized version of the basic algorithm where negative unit clauses are tested on literals in *new* as soon as they are generated as the full-test algorithm described in the next section.

```
M := φ;
D := {A | (true → A) ∈ a set of given clauses};
while D ≠ φ do begin
      D := D − Δ;
      new := CJM_Generator(Δ, M);
      M := M ∪ Δ;
      new' := subsumption(new, M ∪ D);
      if CJM_Tester(new', M ∪ D) ∋ false
            then return(success);
      D := D ∪ new';
end return(fail)
```

Figure 7: Full-test algorithm

4.3.2 Full-Test Algorithm

Figure 7 shows a refined version of the basic algorithm called the full-test algorithm. The algorithm 1) selects Δ from D, 2) performs model extension using Δ and M generating new for the next generation of Δ, 3) performs a subsumption test on new against $M \cup D$, and 4) performs a rejection test on new', which passed the subsumption test, together with $M \cup D$.

Though this refinement seems to be very small on the text level, the complexity of time and space is significantly reduced, as explained later. The points are as follows. The algorithm performs subsumption and rejection tests on all elements of new rather than on Δ, a subset of new generated in the past cycles. As a result, if a falsifying atom [4], X, is included in new, the algorithm can terminate as soon as $false$ is derived from X. That is, the algorithm neither overlooks the falsifying atom nor puts it into D as the basic algorithm does. Thus, it never generates atoms which are superfluous after X is found.

4.3.3 Lazy Algorithm

Figure 8 shows another refinement of the basic algorithm, the lazy algorithm. In this algorithm, it is assumed that two processes, one for generator clauses and the other for tester clauses, run in parallel and communicate with each other.

The tester process 1) requests Δ to the generator process, 2) performs a subsumption test on Δ against $M \cup D$, and 3) performs a rejection test on Δ.

For the generator process,

1) if a buffer, Buf, used for storing a set of atoms which are the results of an application of the model extension rule, is empty, the generator selects an atom, e, from D and sets a code for model extension (**delay** CJM) for e and M onto Buf,

2) waits for a request of Δ from the tester process, and

[4] A falsifying atom, X, is an atom that satisfies the antecedent of a negative clause by itself or in combination with $M \cup D$.

```
process tester:
      repeat forever
            request(generator, Δ);
            Δ' := subsumption(Δ, M ∪ D);
            if CJM_T(Δ', M ∪ D) ∈ false
                  then return(success);
            D := D ∪ Δ'.

process generator:
      repeat forever
            while Buf = φ do begin
                  D := D − {e};
                  Buf :=delayCJM_G({e}, M);
                  M := M ∪ {e} end;
            wait(tester);
            Δ :=forceBuf;
      until D = φ and Buf = φ.
```

Figure 8: Lazy algorithm

3) forces the buffer, Buf, to generate Δ.

delay (above) is an operator which delays the execution of its operand (a function call). Hence, the function call, $CJM_G(\{e\}, M)$, will not be activated during 1), but will be stored in Buf as a code. Later, at 3), when the **force** operator is applied to Buf, the delayed function call is activated. This generates the values that are demanded. Using this mechanism, it is possible to generate only the Δ that is demanded by the tester process. After the required amount of Δ is generated, a delayed function call for generating the rest of the atoms is put into Buf as a continuation.

The atoms are stored in M and D in a way that makes the order of generating and testing the atoms exactly the same as in the basic algorithm. The point of the refinement in the lazy algorithm is, therefore, to equalize the speed of generation and testing while keeping the order of atoms that are generated and tested the same as that of the basic algorithm. This eliminates any excess consumption of time and space due to over-generation of redundant atoms.

4.4 Optimization of Unit Tester Clauses

Given the unit tester clauses in the problem, the three algorithms above can be further optimized. There are two ways to do this.

One is a dynamic way called the lookahead method. In this method, atoms are generated excessively in the generation process in order to apply the rejection rule with unit tester clauses. More precisely, immediately after generating new, the generator process generates new_{next}, which would be regenerated in a succeeding step. Then

new_{next} is tested with unit tester clauses. If the test fails, then new_{next} is discarded whereas new is stored.

$$< \Delta, M > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new$$

$$< new, M \cup D > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new_{next}$$

$$new_{next} \Rightarrow test(A \rightarrow false)$$

The reason why new_{next} is not stored is that testing with unit tester clauses does not require M or D, but can be done with only new_{next} itself. On the other hand, for tester clauses with more than one literal, testing cannot be completed, since testing for combinations of atoms from new_{next} would not be performed.

new_{next} will be regenerated as new in the succeeding step. This means that some conjunctive matching will be performed twice for the identical combination of atoms in a model candidate. However, the increase in computational cost due to this redundancy is negligible compared to the order of total computational cost.

The other method is a static one which uses partial evaluation. This is used to obtain non-unit tester clauses from a unit tester clause and a set of generator clauses by resolving the two.

$$Generator: \ A_1, A_2 \rightarrow C.$$

$$Unit\ tester: \ A \rightarrow false.$$

$$\Downarrow$$

$$Non\text{-}unit\ tester: \ \sigma A_1, \sigma A_2 \rightarrow false.$$

$$where\ \sigma C = \sigma A$$

The computational complexity for conjunctive matching using the partial evaluation method is exactly the same as that using the lookahead method. The partial evaluation method, however, is simpler than the lookahead method, since the former does not need any modification of the prover itself whereas the latter does. Moreover, the partial evaluation method may be able to reduce the search space significantly, since it can provide propagating goal information to generater clauses. However, in general, partial evaluation results in an increase in the number of clauses, Hence it may make performance worse.

The two optimization techniques are equally effective, and will optimize the model generation algorithms to the same order of magnitude when they are applied to unit tester clauses.

4.4.1 Summary of Complexity Analysis

In this section, we briefly describe the time and space complexity of the algorithms described above. The details are discussed in [Hasegawa *et. al.* 92a]. For simplicity, we assumed the following.

1) The problem consists of generator clauses with two antecedent literals and one consequent literal, and tester clauses with at most two literals.

2) Δ is a singleton set of an atom selected from D.

3) The rate at which conjunctive matchings succeed for a generator clause, and atoms generated as the result pass a subsumption test, the survival rate, is $\rho(0 \leq \rho \leq 1)$.

4) The order in which Δ is selected and atoms are generated according to Δ is fixed for all of the three algorithms.

Table 1 summarizes the complexity analysis. T/S/G stands for complexity entry of rejection test /subsumption test/model extension, and M stands for the required memory space. The value of $\alpha(1 \leq \alpha \leq 2)$ represents the efficiency factor of the subsumption test. $\alpha = 1$ means that a subsumption test is performed in a constant order, because the hashing effect is perfect. $\alpha = 2$ means that a subsumption test is performed in a time proportional to the number of elements, perhaps because a linear search was made in the list. As for the condensed detachment problem, the hashing effect is very poor and α is very close to two.

The memory space required for the basic, full-test/lazy and lazy lookahead algorithms decreases along this order by a square root for each. This means that the number of atoms generated decreases as the algorithm changes, which in turn implies that the number of subsumption tests decreases accordingly. In the case of $\alpha = 2$, the most expensive computation of all is a subsumption test, and a decrease in its complexity means a decrease in total complexity. On the other hand, in the case of $\alpha = 1$, the most expensive computation of all is the rejection test with two-literal tester clauses. This situation, however, is the same for all of the algorithms and adopting lazy computation will result in speedup by a constant factor. In any case, by adopting lazy computation, the complexity of the total computation is dominated by that of the rejection test .

4.4.2 Performance Experiment

An experimental result is shown in Table 2. The example, Theorem 4, is taken from [Overbeek 90]. We did not use heuristics such as weighting and sorting, but only limited term size and eliminated tautologies.

Every algorithm is implemented in KL1 and run on a pseudo Multi-PSI in PSI-II [Nakashima and Nakajima 87]. The OTTER entry represents the basic algorithm optimized for unit tester clauses and implemented in KL1. The figures in parentheses are of algorithms for tester clauses with two literals as a result of applying partial evaluation to unit tester clauses. In unify entries,

Table 2: Experimental result (Theorem 4)

		basic	full-test	lazy	lazy lookahead	OTTER
Time (sec)		>14000 (463.86)	409.17 (82.40)	407.58 (81.82)	210.45 (81.69)	409.16 (462.13)
Unify		— (43981+74254)	1656+74800 (43981+4158)	1656+74737 (43981+4158)	81956+4095 (43981+4095)	1656+74800 (43981+74254)
Subsumption test		— (5674)	5736 (596)	5736 (596)	593 (593)	5736 (5674)
Memory	M	— (272)	272 (63)	272 (63)	63 (63)	272 (272)
	D	— (1375)	1384 (209)	1384 (209)	209 (209)	1384 (1375)

Table 1: Summary of complexity analysis

	Unit tester clause			
	T	S	G	M
basic	ρm^2	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	ρm^2	$\mu m^{2\alpha}$	m^2	ρm^2
lazy lookahead	m^2	$(\mu/\rho)m^\alpha$	m/ρ	m

	2-literal tester clause			
	T	S	G	M
basic	$\rho^2 m^4$	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	$\rho^2 m^4$	$\mu m^{2\alpha}$	m^2	ρm^2

† m is the number of elements in model candidate when *false* is detected in the basic algorithm.
‡ ρ is the survival rate of a generated atom, μ is the rate of successful conjunctive matchings ($\rho \leq \mu$), and α is the efficiency factor of a subsumption test.

a figure to the left of + represents the number of conjunctive matchings performed in tester clauses, and a figure to the right of + represents the number of conjunctive matchings performed in generator clauses.

These results are a fair reflection of the complexity analysis shown in Table 1. For instance, to solve Theorem 4 without partial evaluation optimization, the basic algorithm did not reach a goal within 14,000 seconds, whereas the full-test and lazy algorithms reached the goal in about 400 seconds. The most time-consuming computation in all of the three algorithms (basic, full-test and lazy), is rejection testing. The difference in the time complexity between the basic algorithm and the other two algorithms is $(\mu\rho^2 m^{4\alpha})/(\mu m^{2\alpha}) = \rho^2 m^{2\alpha}$, which results in the time difference mentioned above.

The basic algorithm and the full-test/lazy algorithm do not differ in the number of unifications performed in the tester clauses. However, the number of unifications performed in the generator clauses and the number of subsumption tests decreases as we move from the basic algorithm to the full-test and lazy algorithms. The decrease is about one hundredth when partial evaluation is not applied. and about one tenth when it is applied.

By applying lookahead optimization, the lazy algorithm is further improved. Though the lookahead optimization and the partial evaluation optimization are theoretically comparable in their order of improvement, their actual performance is sometimes very different. For Theorem 4, the lazy algorithm optimized with partial evaluation took 81.82 seconds, whereas the same algorithm optimized with lookahead optimization took 210.45 seconds. This difference is caused by the difference in the number of unifications performed in the tester clauses. This is because in the lazy algorithms with lookahead optimization, the generator clause, $p(X), p(e(X,Y)) \rightarrow p(Y)$, generates an atom before the unit tester clause, $p(A) \rightarrow false$ tests the atom. In the same algorithm, with the partial evaluation optimization, the instantiation information of A is propagated to the antecedent of $p(X), p(e(X,A)) \rightarrow false$ and the unification failure can be detected earlier.

Partial evaluation optimization is effective for all the algorithms except OTTER. This is because lookahead optimization, in the OTTER algorithm, is already applied to unit tester clauses, and the algorithm remains the basic one for non-unit tester clauses.

5 Parallelizing MGTP

There are several ways to parallelize the proving process in the MGTP prover.

These are to exploit parallelism in:

- conjunctive matching in the antecedent part,
- subsumption test, and
- case splitting

For ground non-Horn cases, it is sufficient to exploit OR parallelism induced by case splitting. Here we use

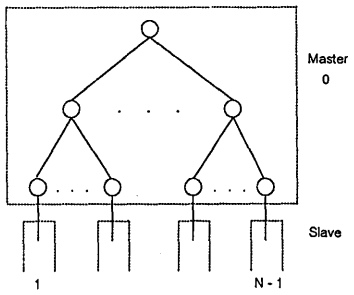


Figure 9: Simple allocation scheme

OR parallelism to seek a multiple model, which produces multiple solutions in parallel.

For Horn clause cases, we have to exploit AND parallelism. The main source of AND parallelism is conjunctive matching. Performing subsumption tests in parallel is also very effective for Horn clause cases.

In the current MGTP, we have not yet considered non-ground and non-Horn cases.

5.1 OR Parallelization for MGTP/G

With the current version of the MGTP/G, we have only attempted to exploit OR parallelism[Fujita and Hasegawa 90] on the Multi-PSI machine[Nakajima *et. al.* 89].

5.1.1 Processor Allocation

The processor allocation methods we have adopted achieve 'bounded-OR' parallelism in the sense that OR-parallel forking in the proving process is suppressed so as to meet restricted resource circumstances.

One simple way of doing this, called *simple allocation*, is depicted in Figure 9. We expanded model candidates, starting with an empty model, using a single master-processor until the number of candidates exceeded the number of available processors. We then distributed the remaining tasks to slave-processors. Each slave processor explored the branches assigned without further distributing tasks to any other processors. This simple allocation scheme for task distribution works fairly well, since the communication cost can be minimized.

5.1.2 Performance of MGTP/G on Multi-PSI

One of the examples we used was the N-queens problem. This problem can be expressed by the following clause set:

Table 3: Performance of MGTP/G on Multi-PSI

	Number of processors				
Problem	1	2	4	8	16
4-queens					
Time (msec)	40	40	39	44	44
Speedup	1.00	1.00	1.02	0.90	0.90
Kred	1.45	1.47	1.48	1.50	1.50
6-queens					
Time (msec)	650	407	266	189	154
Speedup	1.00	1.59	2.44	3.44	4.22
Kred	23.7	23.7	23.7	23.8	23.8
8-queens					
Time (msec)	12,538	6,425	3,336	1,815	1,005
Speedup	1.00	1.95	3.76	6.91	12.5
Kred	460	460	460	460	460
10-queens					
Time (msec)	315,498	159,881	79,921	40,852	21,820
Speedup	1.00	1.97	3.94	7.72	14.5
Kred	11,117	11,117	11,117	11,117	11,117

$$true \rightarrow p(1,1); p(1,2); \ldots; p(1,n).$$
$$true \rightarrow p(2,1); p(2,2); \ldots; p(2,n).$$
$$\ldots$$
$$true \rightarrow p(n,1); p(n,2); \ldots; p(n,n).$$
$$p(X_1,Y_1), p(X_2,Y_2), unsafe(X_1,Y_1,X_2,Y_2) \rightarrow false.$$

The first N clauses simply express every possibility of placing queens on the N by N chess board. The last clause expresses the constraint that a pair of queens must satisfy. The problem can be solved when either a model (one solution) or all of the models (all solutions)[5] are obtained for the clause set.

Performance was measured on the MGTP/G prover running on the Multi-PSI with the simple allocation method. Table 3 gives the result of the all-solution search on the N-queens problem. Here we should note that the total number of reductions stays almost constant, even though the number of processors used increases. This means that no extra computation is introduced by distributing tasks. Speedup obtained by using up to 16 processors is shown in Figures 10 and 11. For the 10-queens and 7-pigeons problems, the speedup obtained as the number of processors increases is almost linear. The speedup rate is small only for the 4-queens problem. This is probably because the constant amount of interpretation overhead in such a small problem will dominate the tasks required for the proving process.

[5]All models can be obtained, if they are finite, by the MGTP interpreter in all-solution mode.

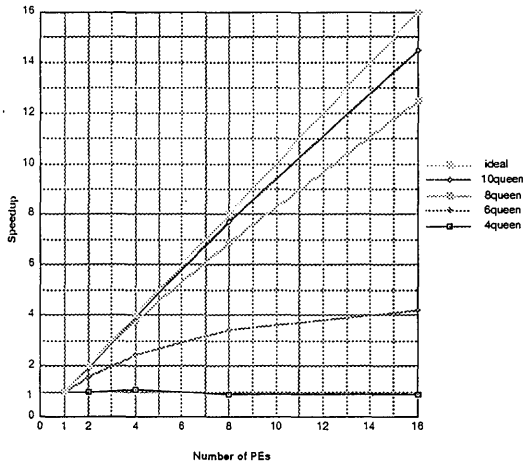


Figure 10: Speedup of MGTP/G on Multi-PSI (N-queens)

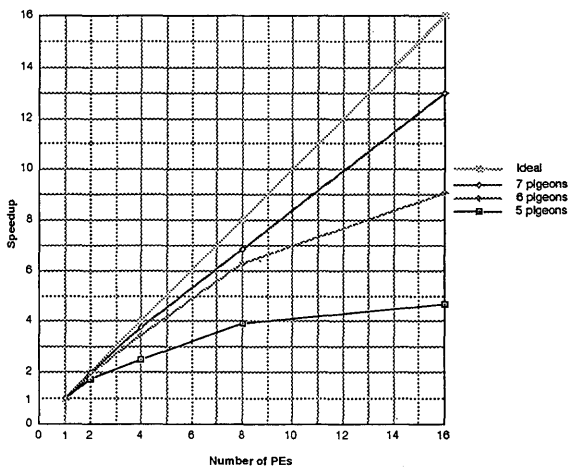


Figure 11: Speedup of MGTP/G on Multi-PSI (Pigeon hole)

5.2 AND Parallelization for MGTP/N

We have several choices when parallelizing model-generation based theorem provers:

1) proofs which change or remain unchanged according to the number of PEs used,

2) model sharing (copying in a distributed memory architecture) or model distribution, and

3) master-slave or masterless.

The proof obtained by a proof changing prover may be changed according to a change in the number of PEs. We might get super-linear speedup if the length of a proof depended on the number of PEs used. However, we cannot always expect an increase in speed as the number of PEs increases.

On the other hand, a proof unchanging prover does not change the length of the proof, no matter how many PEs we use. Hence, we could always expect greater speedup as the number of PEs increased, though we would only get linear speedup at best.

With model sharing, each PE has a copy of the model candidates and distributed model-extending candidates. With model distribution, both the model candidates and model-extending candidates are distributed to each PE.

Model sharing and model distribution both have advantages and disadvantages. From the distributive processing point of view, with model distribution, we can obtain memory scalability and more parallelism than with the model sharing method. For a newly created atom δ, there are n parallelisms in the model distribution method, since we can perform conjunctive matchings and subsumption tests for it in parallel where n is the number of processors. On the other hand, in the model sharing method, we cannot exploit this kind of parallelism for a single created atom unless conjunctive matchings and subsumption tests are made for a different region of model candidates.

From the communication point of view, however, the communication cost with model sharing is less than with model distribution. The communication cost with model distribution increases as the number of PEs increases, since generated atoms need to flow to all PEs for subsumption testing. For example, if the size of model elements finally obtained is M, the number of communications amounts to $O(M^2)$ for a clause having two antecedent literals. On the other hand, with model sharing, we do not have to flow the generated atoms to all PEs. In this case, time-consuming subsumption tests and conjunctive matchings can be performed independently at each PE, with minimal inter-PE communication.

The master-slave configuration makes it easy to build a parallel system by simply connecting a sequential version of MGTP/N on a slave PE to the master PE. However, its devices must be designed to minimize the load on

the master process. On the other hand, a masterless configuration such as ring connection allows us to achieve pipeline effects with better load balancing, whereas it becomes harder to implement suitable control to manage collaborative work among PEs.

Our policy in developing parallel theorem provers is that we should distinguish between the speedup effect caused by parallelization and the search-pruning effect caused by strategies. In proof changing parallelization, changing the number of PEs is merely betting, and may cause a strategy to be changed for the worse even if it results in the finding of a shorter proof.

In order to ensure the validity of our policy, we implemented proof changing and unchanging versions. In the following sections, we describe actual parallel implementations and compare them.

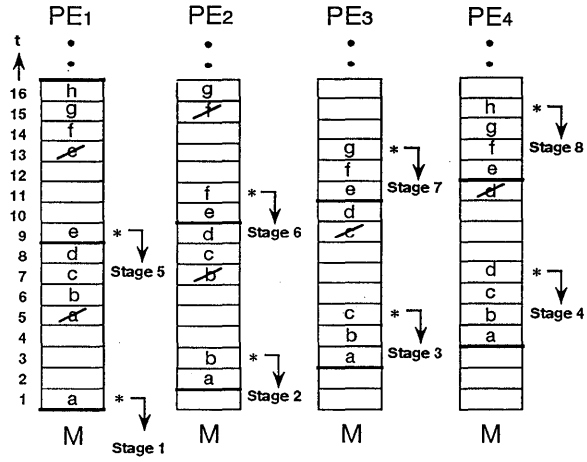


Figure 12: Proof Changing and Model Sharing

5.2.1 Proof Changing Implementation

1. Model Sharing
   This implementation uses model sharing, and a ring architecture in which $process_i (1 \leq i < n)$ is connected to $process_{i+1}$ and $process_n$ is connected to $process_1$, where n is the number of PEs[Hasegawa 91a].

   $process_i$ has a copy of model candidates M and distributed model-extending candidates D_i.

   A rough sketch of operations performed in $process_i (1 < i \leq n)$ follows.

   (1) Receive Δ_{i-1} from $process_{i-1}$.

   (2) Pick up an atom δ_i from D_i such that δ_i is not subsumed by any elements in M and Δ_{i-1}. $D_i := D_i - \{\delta_i\}$.

   (3) $\Delta_i := \Delta_{i-1} \cup \{\delta_i\}$.

   (4) If $CJM_{Tester}(\{\delta_i\}, M \cup \Delta_{i-1}) \ni false$ then send a termination message to all processes, otherwise,

   (5) $D_i := D_i \cup CJM_{Generator}(\{\delta_i\}, M \cup \Delta_{i-1})$.

   (6) $M := M \cup \Delta_i$ (update M in $process_i$).

   (7) Send Δ_i to $process_{i+1}$.

   For $process_1$, instead of actions (3) and (6), the following actions are performed.

   (3') $\Delta_1 := \{\delta_1\}$, and

   (6') $M := M \cup \Delta_n$.

   Note that actions (4)~(8) can be performed in parallel.

   Figure 12 shows how models are copied, and conjunctive matching is executed in a pipeline manner in the case of $n = 4$.

   A letter denotes a model candidate element and an asterisk indicates an element on which conjunctive matching is performed. For example, $process_1$ on PE_1 selects an unsubsumed model element a (from its own model-extending candidate) at time t_1, and sends it to $process_2$ on PE_2.

   $process_2$ stores element a into the model candidates in PE_2, proposes a model-extending element b, sends a and b to the $process_3$, and starts conjunctive matching of b and $\{a\} \cup M$.

   Note that conjunctive matching in a $process_i$ can be overlapped. For example, the conjunctive matching in stage 6 does not have to wait for the completion of the conjunctive matching in stage 2. This exploits pipeline effects very well, resulting in low communication cost compared to the computation cost for conjunctive matching.

2. Model Distribution
   This implementation takes model distribution and a ring architecture. Each process has its own distributed model candidates and distributed model-extending candidates. The algorithm for each process is similar to the sequential basic algorithm. They differ in that: 1) conjunctive matching cannot be completed in one process because model candidates are distributed. Thus the continuations of conjunctive matching in each process need to go around the ring, and 2) newly created atoms have to go around the ring for subsumption testing.

5.2.2 Proof Unchanging Implementation

We implemented a proof unchanging version in a master-slave configuration, and model sharing based on the lazy model generation. In this implementation, generator and subsumption processes run in a demand-driven mode,

while tester processes run in a data-driven mode. The main advantages of this implementation are as follows:

1) Proof unchanging allows us to obtain greater speedup as the number of PEs increases.

2) By utilizing the synchronization mechanism supported by KL1, sequentiality in subsumption testing is minimized.

3) Since slave processes spontaneously obtain tasks from the master, and the size of each task is well equalized, good load balancing is achieved.

4) By utilizing the KL1 stream data type, demand-driven control is easily and efficiently implemented.

By using demand-driven control, we cannot only suppress unnecessary model extensions and subsumption tests but also maintain a high running rate, which is the key to achieving linear speedup.

The model generation method consists of three tasks:

1) generation,

2) subsumption test, and

3) rejection test.

We provided three processes to cope with this:

- G(generator),
- S(subsumption tester), and
- T(rejection tester).

The $G/T/S$ process has a pointer $i/j/k$ which indicates an element of the stack, shown in Figure 13. The stack elements are model candidates or model-extending candidates. In the figure, M denotes model candidates for which conjunctive matching performed by G is completed and D denotes model-extending candidates on which the subsumption test is completed. $G/T/S$ process iterates the following actions.

G: performs model extensions by using the i-th element (Δ) and the $1,\ldots,i-1$ -th elements (M), and sends newly created atoms to S. $i := i+1$.

S: performs subsumption tests on the newly created atoms against $1,\ldots,k-1$ -th elements ($M \cup D$), and pushes the unsubsumed atoms to the stack. $k := k+l$ where l is the number of unsubsumed atoms.

T: performs model rejection tests on the j-th element and the $1,\ldots,j-1$ -th elements.

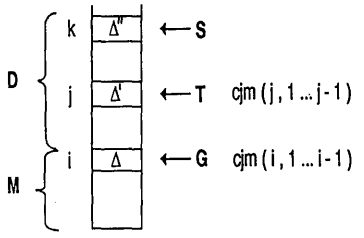


Figure 13: Lazy Implementation

Figure 14 shows a process structure for the proof unchanging parallel implementation. The central box represents the shared model and model-extending candidates.

The upper boxes represent atoms generated by the generator G_i and the arrows indicate the order in which the atoms are sent to the master process. Proof unchanging is realized by keeping this order. To make the system proof unchanging, the sequence order in which M and D are updated must remain the same as the sequence in a sequential case. The master process sends an atom generated by a generator process to a subsumption tester process in the same order as the master receives the atom, that is, the master aligns the elements generated by generator processes so as to be in the same order as in the sequential case.

Many $G/T/S$ processes work simultaneously. The master process is introduced to control task distribution, that is, giving a different task (Δ) to a different process. Each S process requests Δ'' to a G process through the master process. This means that the communication between G and S processes is indirect.

The critical resource for S processes is the model-extending candidates D. The critical regions are the updating of D by $D := D \cup new'$ and a part of $subsumption(new, M \cup D)$ (see Figure 8).

Most elements of $M \cup D$ have already been determined by some subsumption tester process and synchronization in subsumption testing can be minimized so that most parts of subsumption tests should not be critical.

To exclusively access the critical resource D, each S process requests to the master a pair of Δ'' and a key which indicates the right to update. If Δ'' is subsumed by the already determined elements in $M \cup D$, the key is returned to the master process without any reference to the key. In this case, there is no synchronization with other S processes. If Δ'' is not subsumed by the already determined elements in $M \cup D$, the S process refers to the key to see if it has the right to update, and updates D by $D := D \cup \Delta''$ if it has. Otherwise, the process waits until the other S process updates D. If the other S process updates D, the subsumption test is performed on the added elements.

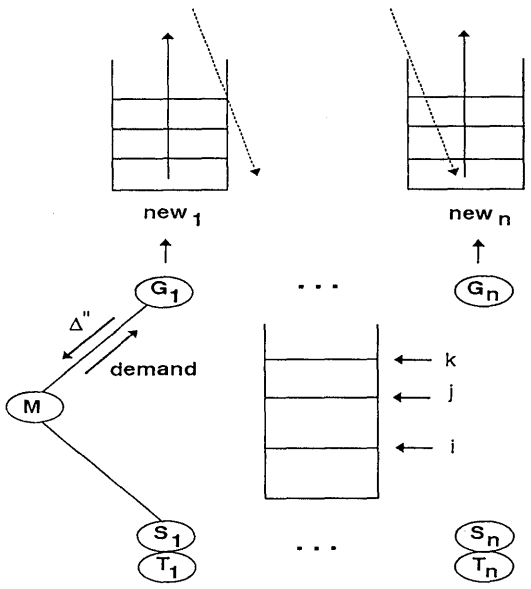


Figure 14: Proof Unchanging

The critical resources for the G processes are both the model candidates M and the model-extending candidates D. This is similar to tester processing.

5.2.3 Performance of MGTP/N on Multi-PSI and PIM

Some experimental results for the proof changing and unchanging versions in model sharing are shown in Tables 4 and 5, and Figures 15 and 16. Each program is implemented in KL1 and runs on the Multi-PSI.

Table 4 shows a performance comparison between the two versions with 16 PEs. In the proof unchanging version (PU column), we limited the term size and eliminated tautologies. In addition to the above, in the proof changing version (PC column), we used heuristics such as weighting and sorting. All problems are condensed detachment problems [McCune and Wos 91].

We measured performance with 1, 2, 4, 8 and 16 PEs. In the PC time entry column, the number of PEs in parentheses indicates the number of PEs which yield the best performance. In the proof unchanging version, we always got the best performance with 16 PEs, whereas we sometimes got the best performance with 8 PEs in the proof changing version. We also have an example in which we got the best performance with 2 PEs.

This comparison implies that super-linear speedup does not always signify an advantage in a parallelization method, because the proof unchanging version always beats the proof changing version in absolute speed with the problems used in the table.

Figures 15 and 16 display the speedup ratio for the problems #3, #58, #77, #66, #92, and #112 using the

Table 4: Performance Comparison (16PEs)

Problem		PU	PC
#3	Time (sec)	218.77	6766 (16 PEs)
	KRPS/PE	34.68	25.99
	Speedup	13.27	-
#6	Time (sec)	3.75	157.63 (16 PEs)
	KRPS/PE	12.47	17.75
	Speedup	3.65	6.75
#56	Time (sec)	3.53	10.37 (8 PEs)
	KRPS/PE	13.39	3.97
	Speedup	3.53	415.57
#58	Time (sec)	12.80	27.32 (16 PEs)
	KRPS/PE	27.51	3.75
	Speedup	9.23	66.32
#63	Time (sec)	4.56	48.37 (16 PEs)
	KRPS/PE	20.01	15.24
	Speedup	6.06	11.07
#69	Time (sec)	6.07	23.41 (16 PEs)
	KRPS/PE	16.69	4.52
	Speedup	4.98	2.90
#72	Time (sec)	3.62	12.17 (16 PEs)
	KRPS/PE	14.02	2.10
	Speedup	4.47	45.51
#77	Time (sec)	37.10	62.07 (8 PEs)
	KRPS/PE	36.66	25.62
	Speedup	12.65	109.24

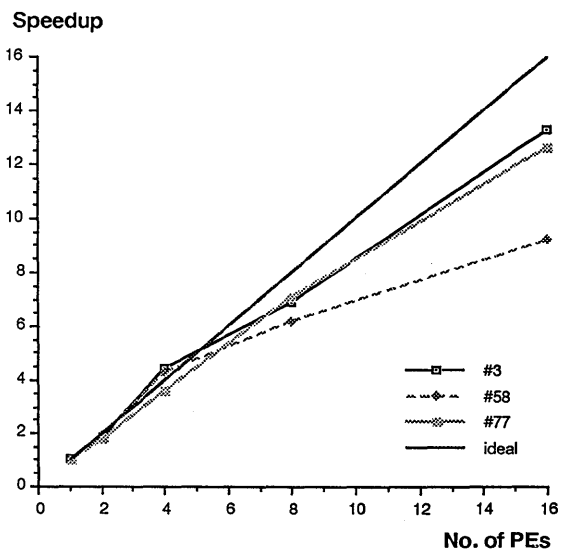


Figure 15: Speedup ratio I

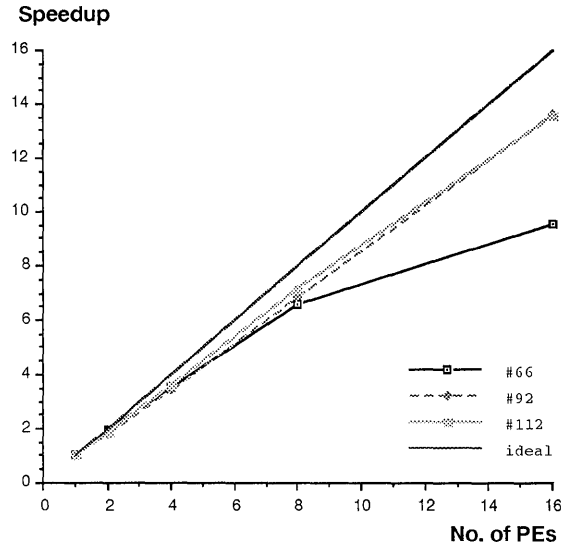


Figure 16: Speedup ratio II

Table 5: Performance for 16/64 PEs

Problem		16 PEs	64 PEs
Th 5	Time (sec)	41725.98	11056.12
	Reductions	38070940558	40759689419
	KRPS/PE	57.03	57.60
	Speedup	1.00	3.77
Th 7	Time (sec)	48629.93	13514.47
	Reductions	31281211417	37407531427
	KRPS/PE	40.20	43.25
	Speedup	1.00	3.60

proof unchanging version. There is no saturation in performance up to 16 PEs and greater speedup is obtained for the problems which consume more time.

Table 5 shows the performance obtained by running the proof unchanging version for Theorems 5 and 7 [Overbeek 90] on Multi-PSI with 64 PEs. We did not use heuristics such as sorting, but merely limited term size and eliminated tautologies. Note that the average running rate per PE for 64 PEs is actually a little higher than that for 16 PEs. With this and other results, we were able to obtain almost linear speedup.

Recently we obtained a proof of Theorem 5 on PIM/m [Nakashima *et. al.* 92] with 127 PEs in 2870.62 sec and nearly 44 billion reductions[6] (thus 120 KRPS/PE). Taking into account the fact that the PIM/m CPU is about twice as fast as the Multi-PSI CPU, we found that near-linear speedup can be achieved, at least up to 128 PEs.

[6]The exact figure was 43,939,240,329 reductions

6 Conclusion

We have presented two versions of the model-generation theorem prover MGTP implemented in KL1: MGTP/G for ground models and MGTP/N for non-ground models. We evaluated their performance on the distributed memory multi-processors Multi-PSI and PIM.

When dealing with range-restricted problems in model-generation theorem provers, we only need matching rather than full unification, and can make full use of the language features of KL1, thereby achieving good efficiency.

The key techniques for implementing MGTP/G in KL1 are as follows:

(1) A given set of input clauses of implicational form are compiled into a corresponding set of KL1 clauses.

(2) Generated models are held by the prover program instead of being asserted.

(3) Conjunctive matching of the antecedent literals of an input clause against a model element is performed by very fast KL1 head unification.

(4) Searching for a model element that matches the antecedent is performed by computing a repeated combination of model elements by means of loop executions instead of backtracking.

(5) Fresh variables for a different instance of the antecedent literal are obtained automatically just by calling a KL1 clause.

These techniques are very simple and straightforward yet effective.

For solving non-range-restricted problems, however, we cannot use the above techniques developed for MGTP/G. If the given problem is Horn, it can be solved by the MGTP prover extended by incorporating unification with occurrence check, without changing the basic structure of the prover. For non-Horn problems, however, substantial changes in the structure of the prover would be required in order to manage shared variables appearing in the consequent literals of a clause. Accordingly, we restricted MGTP/N to Horn problems, and developed a set of KL1 meta-programming tools called the Meta-Library to support full unification and the other functions for variable management.

To improve the efficiency of the MGTP provers, we developed RAMS, MERC, and Δ-M methods that enable us to avoid redundant computations in conjunctive matching. We have obtained good performance results by using these methods on the PSI.

Moreover, it is important to avoid very great increases in the amount of time and space consumed when proving hard theorems which require deep inferences. For this we proposed the lazy model generation method, which

can decrease the time and space complexity of the basic algorithm by orders of magnitude. Experimental results show that significant amounts of computation and memory can be saved by using the lazy algorithm.

The parallelization of MGTP is one of the most important issues in our research project.

For non-Horn ground problems, a lot of OR parallelism caused by case splitting can be expected. This kind of problem is well-suited to a local memory multi-processor such as Multi-PSI, on which it is necessary to make the granuality as large as possible so that communication costs can be minimized. We obtained an almost linear speedup for the n-queens, pigeon hole, and other problems on Multi-PSI, using a simple allocation scheme for task distribution.

For Horn problems, on the other hand, we had to exploit the AND parallelism inherent in conjunctive matching and subsumption. Though the parallelism is large enough, it seemed rather harder to exploit than OR parallelism, since the Multi-PSI is not suited to this kind of fine-grained parallelism. Nevertheless, we found that we could obtain good performance and scalability by using the AND parallelization methods mentioned in this paper.

In particular, the recent results obtained by running the MGTP/N prover on PIM/m showed that we could achieve linear speedup for condensed detachment problems, at least up to 128 PEs. The key technique is the lazy model generation method, that avoids the unnecessary computation and use of memory space while maintaining a high running rate.

For MGTP/N, full unification is written in KL1, which is thirty to one hundred times slower than that written in C on SUN/3s and SPARCs. To further improve the performance of MGTP/N, we need to incorporate built-in firmware functions for supporting full unification, or to develop KL1 compiling techniques for non-ground models.

Through the development of MGTP provers, we confirmed that KL1 is a powerful tool for the rapid prototyping of concurrent systems, and that parallel automated reasoning systems can be easily and effectively built on the parallel inference machine, PIM.

Acknowledgment

We would like to thank Dr. Kazuhiro Fuchi, the director of ICOT, and Dr. Koichi Furukawa, the deputy director of ICOT, for giving us the opportunity to do this research and for their helpful comments. Many fruitful discussions took place at the PTP Working Group meeting. Thanks are also due to Prof. Fumio Mizoguchi of the Science University of Tokyo, who chaired PTP-WG, and many people at the cooperating manufacturers in charge of the joint research.

References

[Bibel 86] W. Bibel, *Automated Theorem Proving,* Vieweg, 1986.

[Bose *et. al.* 89] S. Bose, E. M. Clarke, D. E. Long and S. Michaylov, PARTHENON: A Parallel Theorem Prover for Non-Horn Clauses in *Proc. of 4th Annual Symp. on Logic in Computer Science*, 1989.

[Chikayama *et. al.* 88] T. Chikayama, H. Sato and T. Miyazaki, Overview of the Parallel Inference Machine Operating System (PIMOS), in *Proc. of FGCS'88*, 1988.

[Fuchi 90] K. Fuchi, Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Fujita and Hasegawa 90] H. Fujita and R. Hasegawa, Implementing A Parallel Theorem Prover in KL1, in *Proc. of KL1 Programming Workshop '90,* pp.140-149, 1990 (in Japanese).

[Fujita *et. al.* 90] H. Fujita, M. Koshimura, T. Kawamura, M. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1, *Joint US-Japan Workshop,* 1990.

[Fujita and Hasegawa 91] H. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm, In *Proc. of the Eighth International Conference on Logic Programming,* The MIT Press, 1991.

[Hasegawa *et. al.* 90a] R. Hasegawa, H. Fujita and M. Fujita, A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, *Italy-Japan-Sweden Workshop*, ICOT-TR-588, 1990.

[Hasegawa *et. al.* 90b] R. Hasegawa, T. Kawamura, M. Fujita, H. Fujita and M. Koshimura, MGTP: A Hyper-Matching Model-Generation Theorem Prover with Ramified Stacks, *Joint UK-Japan Workshop,* 1990.

[Hasegawa 91a] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan, In Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[Hasegawa 91b]
R. Hasegawa, A Parallel Model-Generation Theorem Prover in KL1, Workshop on Parallel Processing for AI,IJCAI'91,1991.

[Hasegawa 91c] R. Hasegawa, A Parallel Model Generation Theorem Prover with Ramified Term-Indexing, Joint France-Japan Workshop,Rennes,1991.

[Hasegawa 91d] R. Hasegawa, A Lazy Model-Generation Theorem Prover and Its Parallelization, Joint Germany-Japan Workshop on Theorem Proving, GMD,Bonn,1991.

[Hasegawa *et. al.* 92a] R. Hasegawa, M. Koshimura and H. Fujita, Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers, ICOT-TR-751, 1992.

[Hasegawa *et. al.* 92b] R. Hasegawa, M. Koshimura and H. Fujita, MGTP: A Parallel Theorem Prover Based on Lazy Model Generation, To appear in *Proc. of CADE 92 (System Abstract)*, 1992.

[Koshimura *et. al.* 90] M. Koshimura, H. Fujita and R. Hasegawa, Meta-Programming in KL1, ICOT-TR-623, 1990 (in Japanese).

[Loveland 78] D. W. Loveland, *Automated Theorem Proving: A Logical Basis,* North-Holland, 1978.

[Manthey and Bry 88] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, In *Proc. of CADE 88, Argonne, Illinois,* 1988.

[McCune 90] W. W. McCune, OTTER 2.0 Users Guide, Argonne National Laboratory, 1990.

[McCune and Wos 91] W. W. McCune and L. Wos, Experiments in Automated Deduction with Condensed Detachment, Argonne National Laboratory, 1991.

[Nakajima *et. al.* 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama, Distributed Implementation of KL1 on the Multi-PSI/V2, in *Proc. of 6th ICLP*, 1989.

[Nakashima and Nakajima 87] H. Nakashima and K. Nakajima, Hardware architecture of the sequential inference machine PSI-II, In *Proc. of 1987 Symposium on Logic Programming,* Computer Society Press of the IEEE,1987.

[Nakashima *et. al.* 92] H. Nakashima, K. Nakajima, S. Kondoh, Y. Takeda and K. Masuda, Architecture and Implementation of PIM/m, In *Proc. of FGCS'92*, 1992.

[Overbeek 90] R. Overbeek, Challenge Problems, (private communication) 1990.

[Slaney and Lusk 91] J. K. Slaney and E. L. Lusk, Parallelizing the Closure Computation in Automated Deduction, In *Proc. of CADE 90,* 1990.

[Schumann 89] J. Schumann, SETHEO: User's Manual, Technische Universität München, 1989.

[Stickel 88] M. E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, In *Journal of Automated Reasoning,* **4**:353-380, 1988.

[Stickel 89] M. E. Stickel, The Path-indexing method for indexing terms, Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1989.

[Wos *et. al.* 84] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications,* Prentice-Hall, 1984.

[Wos 88] L. Wos, *Automated Reasoning – 33 Basic Research Problems –,* Prentice-Hall, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

On A Grammar Formalism, Knowledge Bases and Tools for Natural Language Processing in Logic Programming

SANO, Hiroshi and FUKUMOTO, Fumiyo
Institute for New Generation Technology (ICOT)
Sixth Research Laboratory
sano@icot.or.jp fukumoto@icot.or.jp

Abstract

This paper gives an overview of Natural Language Processing (NLP) by adopting the framework of logic programming in ICOT. First, we introduce a grammar formalism called SFTB, a new grammar formalism which has been evolved from the latest research work on contemporary Japanese. SFTB was designed and developed following the outcome of this research and incorporates computational features. Two grammar studies are in current use at the laboratory. One, Localized Unification Grammar (LUG), is based on the phrase-base approach. Another, Restricted Dependency Grammar (RDG), belongs to the family of dependency grammars. Computer-based dictionaries should be thought of as knowledge bases. We have built a dictionary, in the form of LUG, which is available for sentence processing. In addition to the hand-built database, we have developed computer-based dictionaries. Finally, a tool for developing grammar rules which run on a computer has been introduced. Basic grammar rules, described in the LUG form, have been made using the tool. The tool makes it possible to extend basic grammar rules in order to create adequate grammar rules for user applications. We believe that this set of tools is applicable to a well-integrated NLP system as a whole. Readers who are interested in NLP systems that are not described here should refer to [1].

1 Introduction

In this paper, we describe NLP research activities conducted at ICOTs sixth laboratory. The overall goal is to provide a set of power tools that use NLP and consist of (1) a framework of grammar structures for a language (Japanese), (2) grammar formalisms for writing grammar structures for computational use, (3) non-trivial-sized grammar rules running on a computer, (4) computer-based dictionaries that help build dictionary entries and can be used in grammar rules, (5) tools for analysis sentences, such as a syntactic parser, morphological analyzer, grammar writer's workbench, and a dictionary editor. The power tools contain the results of our research activities on NLP through the underlying logic programming and may be thought of as a well-integrated NLP system.

One of the major problems in the area of NLP is an essential lack of cooperation by the power tools with each other and subsequent shared data, tools and systems. Because of the wide variety of (1) grammar formalisms, (2) parsing mechanisms which are independent developments and (3) forms in which dictionary entries are written, most researchers develop parsing systems individually, write several grammar rules and construct dictionaries as they go. If the many tools for computational linguistics described above can be made available in common, we will be able to make progress through data sharing. To develop a well-integrated NLP system, we have conducted the following research.

A Grammar Formalism

First, we should clarify what we mean by *sentence*. To do so we introduce a grammar formalism which provides a criterion for defining a relationship between the meaning of sentences and a sequence of words. The grammar formalism described here is called SFTB. The underlying framework draws inspiration from Japanese language morphology [10] and from the latest research work on contemporary Japanese such as Japanese Phrase Structure Grammar (JPSG)[3]. The framework is intended for use in computational grammar.

Two Grammar Descriptive Frameworks

The next objective is to investigate a grammar descriptive framework to hand-code grammar rules as linguistic information supplied by means of SFTB. These rules should be applicable to computer processing in the framework of logic programming. There are two grammar structures in current use at the laboratory. One, Localized Unification Grammar (LUG), is based on the phrase-base approach and aims at unification based formalisms. Another, Restricted Dependency Grammar (RDG), belongs to the family of dependency grammars and processes sentences in a traditional way.

Dictionaries

We shall describe three kinds of dictionaries. Lexical information used in a computer was required in an implementation of LUG's grammar. This is characterized as the finite syntactic information of attribute value pairs and results in a hand-built dictionary. Although the computer resident dictionary, consisting of about 7,000 entries, is hand-coded, the lexical database for morphological analysis created from existing computer-based resources by a conversion program consists of 150,000 entries. We have a dictionary where each entry has a large amount of syntactic information and sense-related semantic information. It may be thought of as a linguistic knowledge base.

A Tool

Finally, we introduce a grammar rule development system called LINGUIST. LINGUIST consists of a bottom-up parser (BUP-translator[8]) and debugging facilities with a cooperative response coordinator for user interfaces. The system is being integrated into an environment of support tools for developing, testing, and debugging grammar rules written in LUG. With this system, we have developed the basic grammar, which has 800 grammar rules, for contemporary Japanese language including morphological analysis rules.

For academic use, tapes are obtainable free from ICOT. These tapes serve as the linguistic tools mentioned above.

2 SFTB – A Grammar formalism for the Japanese language

The aim of developing SFTB is to investigate linguistic frameworks for computational processing of the Japanese language. This framework is a necessary grammatical basis for processing Japanese sentences by computer. A grammatical basis for a language should provide a concrete and coherent framework for relating the linguistic form and the content conveyed by sentence expression. A computer must operate on the information structure expressing the content of a sentence produced by the framework.

In the SFTB framework, the syntactic structure derived from the linguistic form is based on a compositional line of sentence analysis, but not on a flat dependency analysis, which is the traditional way of handling sentence structure. It is able to cope with the problems of subject and object omission, ellipsis, interdependence on context and so forth. It may also end structureless and patternless Japanese language confusion caused by the lack of a proper syntactic framework.

2.1 The SFTB grammar system

In our grammar system, the central units will be that of morphemes but not words, which are classified into parts of speech generally. The grammar proposed in this paper has morphemics as a part of its grammar system. This is a grammar system which is rooted in [10] (see the survey).

Morphemes come in several varieties. A basic unit called *goki* stands for a concept where a certain relation holds the state of affairs and the idea that the object belongs in the real (cognitive) universe. Units of this kind can be divided into two types. One of the free forms, *jiritsu-goki* forms words by itself. We call one of the bound forms *ketugou-goki*, since the unit must combine with affix(s) in order to form words.

A unit called *setsuji* performs a grammatical function where states of affairs are linked with certain relations. For example, verb endings are a type of morpheme. This approach uses a neutral unit *goki* in relation to the grammatical behavior under syntax.

Example 1 shows a phrase structure produced SFTB framework for the phrase "*Uresi sa no taiigen*" ("Expression of happiness" in English). In this case, the word *Uresi-sa* (Happiness) is derived from the *Uresi*(*Keiyou-goki*) and the affix *sa*.

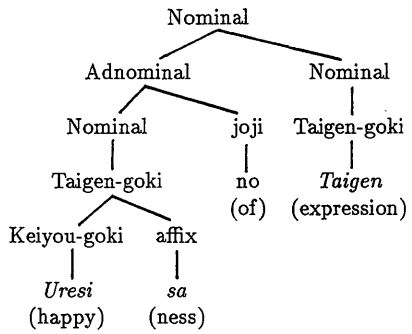


Figure 1: Example of phrase structure

Note that, although the phrase structure is a constituent structure representation from a structural point of view, nodes including feature information are more complex i.e. feature bundles instead of category name. Semantic information is also added to the feature bundles. To make this approach applicable to sentence analysis, all that needs to be done is to integrate morphology and syntax into a comprehensive system.

2.2 Basic patterns and sentence structure

The integration required reconsideration of not only the relation between morphology and syntax but also the general framework of the Japanese language, such as conjugation lists of verbs, word classification , basic sentence patterns and so forth. In this section, we concentrate on the grammar basis for syntactic analysis for Japanese language applicable to computer processing.

First, we should clarify what we mean by a sentence as a criterion. No sentence is uttered or written without the speaker or writer expressing their view on a subject. This may suggest that we ought to extract not only the contents of the sentence but also the intention of the originator from the surface structure of the sentence. Above all, the syntactic forms mapped to verb endings are to be closely related to the meaning of the sentence.

However, this is an ideal case where actual usage depends on context and discourse. As you know, speech is often rambling. One may ask how the content and intuition of graffiti can be extracted. Of course, the criterion mentioned above has to be applied to a limited range of linguistic phenomena. While bearing the above in mind, we offer the criteria and seek sentence structure to map the surface string to an internal representation that is used for NLP by computers.

We characterize the properties of sentence structures we have been investigating, as illustrated in Figure 2.

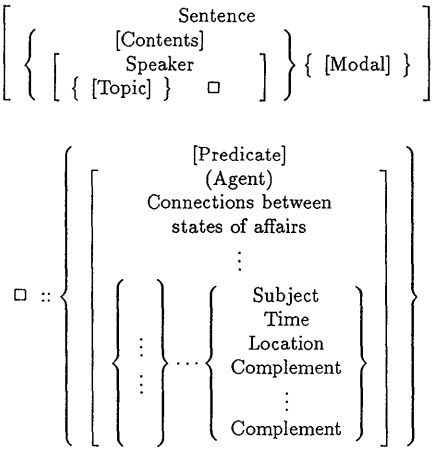


Figure 2: The Sentence Structure

The basic patterns of Japanese sentences are presented in Table 1. The verb "suru", to do, is taken up as a model.

Several types of conjugation lists for verbs are available and the elements of verb endings will be tailored to meet the sentence endings of basic patterns. The lists consist of sentence endings and phrase endings appearing in loosely dependent clauses. The derivation of the conjugation lists illustrated in Table 2 are based on the assumption that verb endings are proportional to the linguistic clues depending on the meaning that the sentence conveys.

Table 1: Basic patterns

Pattern	Sentence endings	Intention
Indicative	*suru/sita*	Neutral
Presumptive	*surudarou/sitadarou*	Presumptive
Volitional	*siyou/sitadarou*	Volitional
Imperative	*siro/suruna*	Imperative

Table 2: Conjugation lists

	Lists (SFTB)		Lists †	
	Form	Type	*Form*	Type
Hana*su*	*su*	Non-perfect indicative	*sa*	1
(To speak)	*sita*	Perfect	*so*	2
	sudaraou	Non-perfect presumptive	*si*	3
	sitadarou	Perfect presumptive	*su*	4
	sou	Positive volitional	*su*	5
	sumai	Negative volitional	*se*	6
	se	Positive imperative	*se*	7
	suna	Negative imperative		
	si	Connective form 1		
	site	Connective form 2		
	seba	Non-perfect conditional		
	sitara	Perfect conditional		
	sitari	Coordinate form		

† School grammar : 1 : Mizen-kei, 2 : Mizen-kei, 3 : Renyou-kei, 4 : Shuusi-kei, 5 : Rentai-kei, 6 : Katei-kei, 7 : Meirei-kei

Compare SFTB's conjugation lists with the grammar lists of the school. The number of verb endings in the school grammar lists is less than seven, in the SFTB's lists, there are over a dozen. For each verb ending form, however, a type name provides sentence patterns, a syntactic function link to the meaning expressed by the sentence.

3 Linguistic Knowledge Bases – Grammar

The well understood way of processing Japanese language is that you read the technical papers and understand the most important part of the framework. Since linguistic knowledge about grammar and dictionaries is described in the natural language itself, there is no straightforward means of applying linguistic knowledge investigated by SFTB to map underlying descrip-

tions onto grammar rules that run on computers. The weakness is in the representations of the programs underlying the parser available for language processing on computers. To solve the problem, we present two computational grammar descriptive frameworks for developing grammar rules built from the SFTB framework. For computer systems with the parser developed in the logic programming framework, these grammar structures give the grammar writer a descriptive framework for writing grammar rules to be used by a parser.

In this section, we describe two computational grammar descriptive frameworks, one is LUG formalism, the other is RDG formalism (RDG), used for writing grammar rules that run on computers. The former is based on the unification grammar that belongs to the phrase structure base grammar. The formalism of the latter takes its stand on the dependency structure grammar.

Much has been done in writing grammar rules to parse natural languages. These grammar rules generally make use of an assumed grammar formalism. Almost no attempt, however, has been made to utilize different ways that base their underlying formalisms on the same grammar for processing a natural language. Thus, although processing methods have been discussed that contrast parsing speeds, required memories and so forth, we have not talked about the merits and the demerits of these grammar structures. In order to ascertain what kind of grammar formalism is appropriate for processing Japanese language, different approaches must be applied to the language.

The original goal of SFTB was to provide a new Japanese language grammar formalism for contemporary Japanese. As applicability to computational linguistics look possible, we are now concentrating on writing grammar rules in terms of LUG and RDG, in the framework of SFTB.

3.1 A Phrase Based Approach – LUG

Definite Clause Grammar (DCG)[6] is one of the bridges connecting NLP and logic programming. Most of our grammar research activities can be regarded as improvements and extensions of DCG. A wide range of parsing techniquea have been suggested based on DCG through underlying context-free grammar. Although the computational effectiveness of DCG is powerful enough to write grammar rules that run directly on a computer, it can be thought of as programming rather than the description of the grammar of a language. It may be said that DCG is less expressive than other grammar formalisms in the sense of mathematical measures. If the process of developing grammar rules is tied to the description of DCG, it will be difficult to develop large grammar structures manual by using the DCG description. To overcome this problem, we have designed LUG to be accessible to grammar writers with little computer experience. Thus, LUG is a grammar specification language designed for users to develop non-trivial grammar expressed in the DCG.

The basic data of LUG is a feature syntax. Categories are expressed as feature sets. Since the feature sets are represented as Prolog lists, the grammar is written in DCG formalisms, allowing users to make use of the BUP[8], BUP-XG[12], SAX[9] translators being developed in the framework of logic programming. As a sample LUG, we present, in Figure 3, an informal representation of the phrase *"Uresi sa no taigen"*.

$$
\left[\begin{array}{ll}
\text{Morph} & \textit{taigen} \\
\text{Category} & \text{Taigen} \\
\text{Marker} & \textit{no} \\
\text{oftype} & \left[\begin{array}{ll}
\text{Morph} & \textit{uresi/sa} \\
\text{Category} & \text{Taigen} \\
\text{Marker} & \textit{sa} \\
\text{Sem} & \text{nominalize}(Y)
\end{array}\right] \\
\text{Sem} & \text{oftype}(X,\text{nominalize}(Y))
\end{array}\right]
$$

Figure 3: LUG for the phrase

The form produced by LUG can be described as a complex constituent that is the result of compositional and functional application. The functional application is used to limit compositional ambiguities caused by unification-oriented structural description. The contextual information of knowledge bases is dealing with the world or pragmatic knowledge about words, for example can be re-unified later. Thus, complete resolution of constituent structures depends on semantic-based and pragmatic-based accounts of subsequent information. With this formalism, the Japanese language grammar is written independently of the task of the application domain.

3.1.1 The Basic Grammars

The LUG formalism has been used to build grammar structures for basic coverage of contemporary Japanese. As of now, grammar structures of 800 grammar rules are usable and are under development for the purpose of increasing coverage.

Remember, however, that the readability of grammar structures is sacrificed when its rules are extended. It is often difficult to keep a large number of grammar rules under control. Even a small loss of attention causing inconsistent grammar can cause ambiguities to increase and analysis to become useless. An important characteristic of the basic grammar structure is that it is orderly divided into 12 groups to the following standards:

- Difficulty in analyzing sentences

  According to Figure 2, a complete analysis of sentences comes from success in understanding the syntactic elements in the structure when in the parsing

process. This is directly and indirectly related to the basic sentence patterns and the sequence that words appear in sentences. The fewer syntactic elements omitted, the easier it is to parse its structure. The greater the difference between the syntactic elements and their corresponding morphemes, the more it costs to analyze a sentence correctly and to grasp its meaning.

Grammar structures can be loosely divided into three levels: elementary, intermediate and advanced levels. We list here some samples of the kinds of grammar structure levels, but are limited to the following.

Elementary level

decision(declaratives), supposition, conjectural form(declaratives), command(imperative), aspect operators, negation, polite form, complements, mood auxiliaries.

Intermediate level

passives, causatives, modal adverbs, spacio-temporal adverbs, topicalized phrases, relatives.

Advanced level

conditional phrases, causal phrases, some connectives, conjunctions and disjunctions of nominal phrases.

3.2 A Dependency Based Approach – RDG

Japanese word order is said to be free. Thus, dependency grammar that only focuses on the relation between two arbitrary constituents as a syntactic structure in a sentence has been well studied. Many NLP systems for the Japanese language have adopted the dependency paradigm as an approach for syntactic analysis.

However, the problem of the dependency structure (it is not a tree but a connected graph structure) which is used in these NLP systems is that useless solutions are generated, which bring about a combinatorial explosion. This is because whether one constituent of a sentence modifies another constituent concerns only the localized information between the two constituents, such as the selectional restriction between a verb and its complements.

In this section, we propose a dependency grammar formalism for the Japanese language called Restricted Dependency Grammar (RDG). A characteristic of RDG is (1) The interpretation of whether one constituent modifies the other or not depends on global information based on the word order of a sentence. So we can suppress the generation of useless solutions. (2) Every constituent of a sentence except the last should modify at least one constituent on its right. So, some linguistic phenomena, themes or ellipses can be treated easily in our approach.

RDG is currently implemented in the SICStus prolog, and is being evaluated by using a Japanese newspaper editorial, with specially attention given to the number of solutions.

In the following subsections, we introduce an outline of RDG formalism, concentrating on the constraint based on the word order of a sentence.

3.2.1 Modifiability rank

A sentence consists of many constituents. We call these phrases. Every phrase of a sentence has two syntactic contrastive aspects, that is, one phrase modifies the other and one phrase is modified by the other. We call these aspects the modifier (henceforth Mer) and the modificand (Mcand). When the Mer of one phrase and the Mcand of another phrase match, we can connect the two phrases by an arc. In RDG formalism, every phrase and arc have a modifiability rank value.

The phrase rank is a classification of a phrase based on the number of Mer and Mcand phrases. For example, a manner adverb such as "yukkuri" (slowly) can modify verbs such as "yomi-nagara" (reading a book), "yomeba" (if you are reading), "yomu-node" (as you read), and "yonda-keredo" (though you read). On the other hand, of all these verbs, a modal adverb, such as "tabun" (probably), can only modify "yonda-keredo" (though you read). This means that the number of Mer "yukkuri" (slowly) is more than that of "tabun" (probably). In a similar way, if a phrase can be modified more than another phrase, the number of Mcand phrases is more than that of the other phrase.

The phrase rank consists of 7 Mer and 7 Mcand ranks. Every phrase has a Mer and Mcand rank. The classification of phrases based on these ranks is given in Table 3.

Table 3: Classification of phrases based on rank

Mer	phrase example
a_1	deictic pronoun
a_2	manner adverb
a_3	noun
a_4	continuous verb
b	condition verb, temporal adverb
c	causal verb
d	contrastive verb, modal adverb

Mcand	phrase example
A_1	deictic pronoun
A_2	manner adverb
A_3	noun
A_4	continuous verb
B	condition verb
C	causal verb
D	contrastive verb

Conditions between phrase ranks are formulated as in (1), and (2).

$$a_2 \succ a_3 \succ a_4 \succ b \succ c \succ d \quad (1)$$

$$a_2 \prec a_3 \prec a_4 \prec B \prec C \prec D \quad (2)$$

(1) shows the Mer rank. (2) shows the Mcand rank. For example, when a manner adverb "yukkuri" (slowly) is classified as $\mathbf{a_2}$, and a modal adverb "tabun" (probably) is classified as **d** from Table 3 Mer Rank, we found that the number of Mer "yukkuri" (slowly) is more than "tabun" (probably) from formula (1).

The arc rank is a classification of an arc based on the phrase's modifiability rank. It is incorporated in the word order of a sentence. We assume that a sentence consists of three phrases, P_i, P_j, and P_k (i < j < k). We can get the dependency structures (Fig 4 (a), and (b)) from this sequence. The arc between P_i and P_j is shown as $\overrightarrow{P_iP_j}$. In Fig 4 (a), we call $\overrightarrow{P_iP_j}$ an adjacent arc of $\overrightarrow{P_jP_k}$. In Fig 4 (b), we call $\overrightarrow{P_jP_k}$ the inside arc of $\overrightarrow{P_iP_k}$.

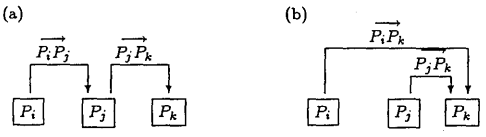


Figure 4: Dependency structure

(1) [Kare-ga] yobu-to heya-kara dete-kita.
When he called ϕ, ϕ went out of the room.
(1)′ Yobu-to [kare-ga] heya-kara dete-kita.
When ϕ called him, he went out of the room.

(1)′ shows a (1) [kare-ga (he)] /[yobu-to (when called)] conversion. We found that the meaning of (1) is different from that of (1)′. When we read sentence (1), we pause on the phrase "yobu-to" (when called). That is, the temporal particle "to" has a function that temporarily disconnects the sentence. So it is hard to say that "kare-ga" (he), which exists before "yobu-to" (when called), is able to modify "dete-kita" (went out of) across a phrase "yobu-to" (when called). This means that there exists a word order constraint between the phrase "kare-ga" (he) and "yobu-to" (when called). Thus, we can get one of the solutions shown in Fig 4 (a). In (1), "Kare-ga" (he) equals P_i, "yobu-to" (when called) equals P_j, and "dete-kita" (went out of) equals P_k.

The rank of an arc incorporates these phenomena (the word order of a sentence). The rank of an arc consists of four levels, corresponding to a phrase's function, that temporarily disconnect the sentence. These four levels are represented by **a**, **b**, **c**, and **d**. These values depend on the values of the Mer and the Mcand rank shown in Table 3. Rank of phrase P_i, P_j, and an arc $\overrightarrow{P_iP_j}$ are shown in Table 4.
In Table 4, the Mer rank of P_i is on the left and the Mcand rank of phrase P_j is on the top. The column shows the rank of an arc $\overrightarrow{P_iP_j}$. A blank column indicates that there is no arc between these two phrases. Conditions between the rank of two arc's is formulated in (3).

Table 4: Rank of P_i, P_j and $\overrightarrow{P_iP_j}$

	A_1	A_2	A_3	A_4	B	C	D
a_1	a						
a_2		a	a	a	a	a	a
a_3			a	a	a	a	a
a_4				a	a	a	a
b					b	b	b
c						c	c
d							d

In (3), for example, when $\mathbf{a} \succ \mathbf{b}$ we say that **b** is lower than **a**.

$$\mathbf{a} \succ \mathbf{b} \succ \mathbf{c} \succ \mathbf{d} \tag{3}$$

Now we show the constraints of word order using the rank of an arc. When the dependency structure is as in Figure 4 (a), the rank between the two arcs should be satisfied by formula (4). When the dependency structure is as in Figure 4 (b), the rank between the two arcs should be satisfied by formula (5).

$$\text{Rank of } \overrightarrow{P_iP_j} \succeq \text{Rank of } \overrightarrow{P_jP_k} \tag{4}$$

$$\text{Rank of } \overrightarrow{P_jP_k} \succeq \text{Rank of } \overrightarrow{P_iP_k} \tag{5}$$

3.2.2 The RDG formalism

In RDG formalism, there are two different kinds of constraint. One is how to make an arc which connects a pair of phrases. The other is whether we can make an arc. These constraints are described as follows.

Structural constraints

1. Absolute dependency
   Every phrase of a sentence except the last should modify at least one phrase on its right. If a phrase modifies a phrase on its right, we can connect them with an arc. The last phrase modifies no other phrase.

2. Crossing
   No two arcs should cross each other.

Linguistic constraints

1. Constraints for phrases
   When P_i and P_j satisfy Table 4, we can get the dependency relation between P_i and P_j. Its column value is the rank value of an arc.

2. Constraints for arcs
   When P_i modifies P_j, (i) the rank of $\overrightarrow{P_iP_j}$ should be lower than the rank of its adjacent arc. (ii) the rank

of $\overrightarrow{P_iP_j}$ should be lower than the rank of its inside arc.

4 Linguistic Knowledge Bases – Dictionaries

Terms used to refer to these dictionaries are (1) computer resident dictionary to be used in syntactic processing by the parser, (2) computer-based dictionary being applicable to applications such as morphological analysis, and (3) machine-tractable dictionary containing lexical information to be interpreted by human readers, i.e. a database used in a data base management system.

The lexical data of these computerized dictionaries includes the following information in (3) only: a list of possible words in a given language (in our research, that language was Japanese), a list of base words and their inflected and derived forms. A classification of semantic information such as word senses.

For dictionary (1), since the lexical data is not necessarily stored in one place, syntactic information about category and the subcategorization behavior of words will appear in grammar rules implicitly.

4.1 Dictionary available in LUG

The dictionary was built by analyzing data from a standard elementary school text. The computer resident dictionary consists of 7,000 entries for each entry in LUG form. Work on the application of the dictionary to the grammar structure has focused on the development of grammar structures written in LUG. Hence, the dictionary relies on our hand-built database and tends to be rather limited in size.

4.2 Dictionary used for Applications

Taking limitations into account, we have developed a program which provides a way of constructing lexical databases through the available machine-readable dictionaries. The lexical database, consisting of 150,000 entries, was created from existing machine-readable dictionaries by using the program. We intend to provide the dictionary as a resource for morphological analysis.

4.3 Dictionary available in DB

A compact but high-capacity computer resident dictionary was extracted from machine-readable dictionaries supplied by IPA [4, 5] by a specialized program. The dictionary can be thought as a linguistic knowledge base and can assist in constructing a restricted specific-domain dictionary for used in NLP, by providing semantic information to the analysis. At present, we utilize the dictionary by using a data base management system.

5 Tools for NLP

We have formalized the SFTB grammar formalism for the Japanese language and grammar rules running on computers are realized in the framework of logic programming using LUG and RDG to demonstrate the descriptive power of their grammar formalisms. On the other hand, we have developed NLP systems for doing computational linguistics with logic programming techniques, such as a dictionary management system, a grammar writer's workbench, a debugging system built around a parser and so forth. In the following section, we introduce a grammar rules development system.

5.1 LINGUIST

LINGUIST is a NLP system with three purposes:

(1) verifying the more detailed nature of the framework of a natural language (Japanese) in a strict enough sense to take an objective view;

(2) developing more useful grammar structures that can be used widely in the domains where the natural language interface to an information retrieval component is used as an intelligent system device; and

(3) having a tool for processing Japanese language and thoroughly trying our grammar ideas.

In the sense of (1), LINGUIST is a grammar development system designed to assist the development of grammar rules expressed in the DCG formalism. LUG, mentioned in Section 3.1 is currently implemented in the LINGUIST, and has been doing well as a development and verification tool in the research of Japanese grammar with respect to computational linguistics. The primary goal in designing the LINGUIST was to efficiently develop grammar rules by computer using logic programming. Thus, the system described with respect to (1) functions as a grammar writer's workbench for NLP.

Once produced, the grammar rules are included in the NLP unit that makes up one part of the user interface of a system, such as an expert system, and the translator of a machine translation system. Then the grammar rules are adjusted to a particular purpose for which the parser applying the grammar to sentence analysis must be able to produce the structure needed by the application domain. Modification of the grammar rules increases with greater application speciality. When this happens, it becomes unclear what the basic grammar rules are.

In order to avoid this problem, LINGUIST has been enhanced with a powerful editing facility that makes it easy to support modification of the grammar rules needed for adjustment to an application domain. With this improvement, the LINGUIST provides users with the

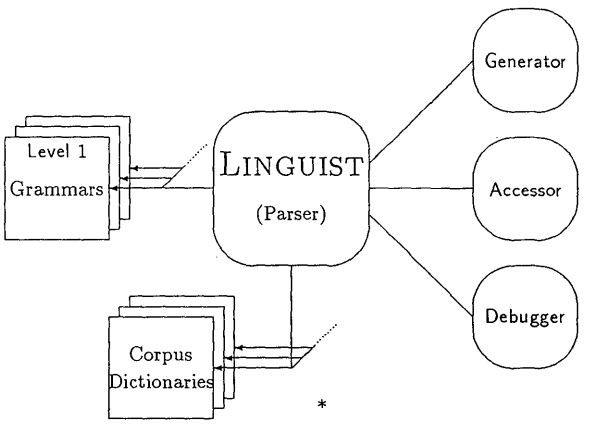
basic grammar rules being developed at ICOT. The application of the basic grammar rules within the LINGUIST simplifies many of the modifications dealt with by the application system builders, as mentioned in (2).

In (3), the system being developed in the framework of logic programming will enable the investigation and processing of various linguistic phenomena in the approach of parsing a natural language (Japanese) by computer.

5.1.1 The Machinery of the LINGUIST

This system initially implemented on PSI-II[1] under ESP[2] is now available on a SPARCstation under CESP[3].

The LINGUIST is organized into three major modules, as illustrated in Figure 5:



Also provided is the basic grammar structures of Level 1~Level 12

Figure 5: The System Configuration

- Generator includes the BUP translator that translates Grammar rules written in DCG into ESP (CESP) code. The user may manually select operations to consult with, save or load a parser program.

- Accessor allows the user to access the parser that is ready to parse and provides inspectors to display the results of parsing. Each inspector provides an iconic menu of operations. Grammar structures can be tested interactively and the results saved on system holders for inspect and on files for future reference.

- Debugger provides a debugging tool for visualizing each invocation during parsing as mouse-sensitive operations. A tracing facility that follows the progress left by the parser is built into the module.

Together these provide a menu-based interface that makes communication with the LINGUIST easy.

5.2 Other applications of the system

As of autumn 1991, the LINGUIST consists of the software itself, a set of grammar rules and a set of dictionaries used by it. The LINGUIST allows an advanced user to extend and modify the basic grammar rules for a specific usage. On the other hand, for inexperienced users the system with its debugging facilities allows the process to be mentioned and helps comprehension of how the grammar rules are applied.

6 Final Remarks

In this paper, we presented an overview of the achievements of our lengthy study in the ICOT project: (1) A framework of Japanese grammar called SFTB, (2) Two grammar formalisms, LUG and RDG, both based on logic programming, (3) Dictionaries thought of as linguistic databases, (4) A grammar rule development system, LINGUIST.

In point (1), our framework of Japanese grammar has the sentence level and structural aspects of Japanese sentence construction. It, also, covers the basic linguistic phenomena of contemporary Japanese and these phenomena are systematically ordered in accordance with the standards, as mentioned in Section 3.1.1.

Section 3 of the paper outlined two grammar formalisms. LUG is a unification-based grammar formalism whose syntactic notation is DCG and is a kind of context-free structure grammar. At present, grammar rules written in LUG formalism on the basis of SFTB are under evaluation by using elementary school texts. On the other hand, RDG is the based on dependency grammar formalism with a mechanism producing the connected graph structures of sentences. We are currently testing its descriptive power by using Japanese newspaper editorials.

In Section 4, we described several kinds of dictionaries thought of as linguistic databases. First, the dictionary consists of about 7,000 entries and is used for analysis as part of the LUG grammar structure. Each entry has the detailed information needed to perform morphological analysis and syntactic parsing. Second, the TRIE structure dictionary has a huge number of dictionary entries built for use in morphological analysis. The dictionary consists of about 150,000 entries.

In Section 5, a tool for the grammar rules development system called LINGUIST was introduced. With

[1]Personal Sequential Inference Machine developed at ICOT.
[2]Extended Self-contained Prolog developed at ICOT.
[3]Common ESP based on ESP developed at AI Language Institute Corp.

LINGUIST we developed basic grammar structures with 800 grammar rules written in LUG formalism. Thus, LINGUIST provides not only an environment for the development of grammar rules but also an environment for modifying grammar rules that will be utilized in various NLP application systems.

The LINGUIST software and the basic grammar rules mentioned above are available to the general public. The dictionaries are also freeing available. We hope these are extensively utilized in various NLP systems.

Availability

Academic users of LINGUIST can obtain free a magnetic tape of the CESP code of the LINGUIST system. The basic grammar rules for the Japanese language in LUG is available as an appendix to the LINGUIST system. The software can be ordered from Mr.Yukio Shigihara, Deputy Chief Research Planning Section, Research Planning Department Research Center, ICOT.

References

[1] Akasaka, K., *et al.* (1989). Language Tool Box (LTB) A Program Library of NLP Tools, *ICOT Technical Report: TR-521, ICOT.*

[2] Fukumoto, Fumiyo., *et al.* (1991). A Framework for Restricted Dependency Grammar, *Papers from 3rd International Workshop on Natural Language Understanding and Logic Programming, pages 68–81.*

[3] Gunji, Takao (1987). Japanese Phrase Structure Grammar A unification-Based Approach, *Studies in Natural Language and Linguistic Theory, D.Reidel Publishing Company.*

[4] Keisankiyou Nihongo Kihon Doushi Jisho IPAL (Basic Verbs) (in Japanese), (1987). *Information Technology Promotion Agency, Japan.*

[5] Keisankiyou Nihongo Kihon Keiyoushi Jisho IPAL (Basic Adjectives) (in Japanese), (1990). *Information Technology Promotion Agency, Japan.*

[6] Pereira, F. & Warren, D (1980). Definite clause grammar for language analysis – a survey of the formalism and a comparison with augmented transition networks, *Artificial Intelligence, Vol.13, No.3, pages 231–278.*

[7] Masuoka, Takashi & Takubo, Yukinori (1989). Kiso Nihongo Bunpou (in Japanese), Tokyo. Kuroshio Shuppan.

[8] Matsumoto, Yuji., *et al.* (1983). BUP : A Bottom-Up Parser Embedded in Prolog, *New Generation Computing, Vol.1, No.2, pages 145–158.*

[9] Matsumoto, Yuji. & Sugimura, Ryouichi (1986). Ronrigata Gengoni motozuku Koubun Kaiseki Sisutemu SAX (in Japanese), *Computer Software, Vol.3, No.2, Japan Society for Software Science and Technology.*

[10] Morioka, Kenji (1987). Goi no Kousei (in Japanese). Tokyo, Meiji Shoin.

[11] Sano, Hiroshi (1990a) Shizen Gengo Jikken Shien Kankyou LINGUIST (in Japanese), *Papers form 8th Symposium on Fifth Generation Computer Technology, ICOT.*

[12] Tokunaga, Takenobu., *et al.* (1988). LangLAB : A Natural Language Analysis System, *Papers from 12th International Conference on Computational Linguistics, Vol.II.*

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Argument Text Generation System (Dulcinea)

IKEDA Teruo, KOTANI Akira†, HAGIWARA Kaoru and KUBO Yukihiro

Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan

Mitsubishi Electric Corporation†
5-1-1, Oofuna, Kamakura, Kanagawa 247, Japan†

Abstract

A generation of texts to justify some opinion requires clear expression of the system's standpoint and beliefs along with the proper strategy for structuring text. We call these kinds of texts *Argument Texts*. This paper is intended to investigate the argument strategy for the coherence of text and definite expression of standpoints. Moreover, we developed the argument text generation system, Dulcinea, using this argument strategy.

This strategy is plausible for the multi-paragraph text generation in a very narrow domain, but we believe that it is one of the answers to the question: "What relations, plans and schemas are necessary to support the planning of coherent multi-paragraph texts?".

The system generates text to justify an argument goal. The text, which reflects the standpoint and the judgment of the system, is represented by an FTS (Functional Text Structure). The FTS represents not only the semantic contents of the argument but the system's standpoint, the judgments and the linguistic constraints.

1 Introduction

Natural language generation systems produce various utterances: from single sentences in a dialog to coherent paragraphs. In recent years, the volume of text generated in text generation research has increased. Many natural language generation systems are able to generate multi-paragraph texts. The quality of these texts is also improving. The center of research is shifting from linguistic realization, which deals with linguistic forms, to structure planning, which produces semantic structures to attain the system's communicative intention. Not only the propositional content, but also the writer's intention and viewpoint are being focused on.

The multi-paragraph texts written by humans are appropriately structured to present their intentions efficiently, and to develop the topics according to their beliefs, interests and viewpoints. These properly structured coherent texts are able to express judgments and attitudes on topics based on the standpoint of the author. By computer, however, it is difficult to produce coherent texts. Therefore, it is necessary to consider coherence and appropriate structure planning for generating high quality multi-paragraph texts by computer.

Especially, the generation of texts to justify some opinion requires clear expression of the system's standpoint and beliefs along with the proper strategy for structuring text. We call these kinds of texts *Argument Texts*. This paper is intended to investigate the argument strategy for the coherence of text and definite expression of standpoints. Moreover, we developed the argument text generation system, Dulcinea, using this argument strategy.

Section 2 illustrates the features of the argument texts and gives a brief description of the belief contents, the plans to generate semantic contents for each constituent of the argument texts, and an abstract text structure form called FTS (Functional Text Structure). Section 3 describes an overview of the Dulcinea argument texts generation system. The system has four processes: generating the semantic contents of arguments, organizing linguistic text structure with argument strategy, sentence level organization for orders and connections, and realizing texts. Section 4 gives an example of argument text generation.

2 What are Argument Texts?

2.1 Features of Argument Texts Written by Humans

An argument text is a set of sentences in support of some opinion. They are generated according to some argument strategies, in order to persuade the reader to agree with the opinion. An argument strategy based on linguistic knowledge is useful to generate effective text to persuade the reader. As a manner of showing the justification of an opinion, for example, some texts give a simple but forceful sentence while others spend paragraphs in explaining the detailed grounds step by step. In addition to how justification is given, it is important to reinforce the argument with related topics. Adding examples to the grounds increases the persuasiveness of a text. Moreover, a technique in which a text mentions an expected

opposing argument and refutes this argument is an effective way of persuasion.

As mentioned above, giving not only the grounds for the conclusion, but also developing topics with examples and opposing arguments persuades the reader more effectively. Such a text must reflect that the writer holds a consistent attitude from a specific standpoint to the topic. An argument text may become vague and not clearly state a view if it does not show a consistent attitude by the writer. Thus it is important to reflect a writer's consistent attitude in natural language expressions by considering the coherence of the text.

2.2 Related Work in Text Generation

In his research, Hovy developed a system[Hovy 1988] that achieves various pragmatic goals to convey more information than that contained in the literal meanings of words. This he did by setting rhetorical goals as intermediate goals between the pragmatic aspects of communication and the syntactic decision of the text generator. As a result, a single semantic content can produce a variety of texts which reflect various conversational settings in various ways.

Hovy's purpose was to connect wide-range pragmatic aspects to natural language expression by various conversational settings and rhetorical goals. His work was successful in this point, but did not give a concrete structuring method to make texts coherent. According to Hovy[Hovy 1990b], one of the unsolved problems in the field of generation by computer is what relations, plans and schemas are necessary to support the planning of coherent multi-paragraph texts. We investigate this problem in the narrow domain of the argument, which we choose as one of the applications of multi-paragraph text generation.

To solve our problems, we investigated the following: what semantic content affects the reader, what text structure should we organize and how should we represent the text structure efficiently. As a result, we described various argument strategies on three levels. First, the plans for generating the semantic content of each constituent of the argument texts. Second, the prescriptive knowledge for organizing the linguistic text structure by combining the constituents. Third, the representation form of the local relations between adjacent sentences that holds within the argument texts.

Many text generation systems employ a model for discourse structure. RST[Mann and Thompson 1987] and Schema[McKeown 1985] are typical examples of a model for generating a coherent discourse structure. Mann and Thompson formalized a set of about 25 relations sufficient to represent the relations between adjacent blocks of text by RST. McKeown's schema represents the structure of stereotypical paragraphs for describing objects, and selecting the proper schema from her four schemas enforces this coherence. The schema that describes the typical format of argument text is suitable for our system's generation process, because it is driven by one global intention (i.e. *insisting the system's standpoint effectively*), and it completes the multi-paragraph text without any interaction with the user. In fact, Dulcinea uses the schema-like knowledge to generate the semantic contents of the text and to organize the text structure.

By schemas, however, it is difficult to represent the local relations between adjacent sentences within the blocks. We represent the relations between adjacent sentences with the RST-like representation form, called FTS (Functional Text Structure). The plans for each constituent of argument text generate the semantic contents and each linguistic structure which is represented by the FTS.

2.3 Generation Process of Dulcinea

The following is a brief review of the generation process of Dulcinea. At first, we set the standpoint of the system by giving it an argument goal. The system's standpoint is whether some states of affairs are good or not good. These are the only possible judgments of the state of affairs. Dulcinea makes the semantic contents of an argument justify the given argument goal according to its beliefs, and represents them with a data structure called the *Argument Graph*. Then, the argument strategy on linguistic text structure is applied to the argument graph to organize an abstract text structure, which is represented by the FTS. The FTS represents not only the semantic content, but also the text structure, according to the standpoint of Dulcinea, and the belief necessary to generate the persuasive argument text. The FTS produces various surface syntactic text structures. Finally, the best text structure is selected and used to form natural language expressions.

The rest of this section describes Dulcinea's belief contents, gives an argument graph for representing semantic contents, and gives the FTS representation form of the text structure.

2.4 Belief Contents and the Argument Goal

Dulcinea's standpoint, which is set by the given argument goal and system beliefs, is the basis of the argument. The beliefs consist of three types of belief contents: *Fact, Rule*,and *Judgment*. Every element of *Fact* is a belief that Dulcinea believes to be true in the real world. Every element of *Rule* is a causal relation between two states of affairs. Every element of *Judgment* is a belief content that the system regards as good or not good. Figure 1 is an example of beliefs.

We give one of the three kinds of modal expressions, defined by the judgments in the table below, for the state

- Rules
  1. enforce[obj=one-way-system, loc=L], enforce[obj=two-way-lane, loc=L, pol=0] ⇒ change[obj2=bus-route, loc=L].
  2. change[obj2=bus-route, loc=L] ⇒ decrease[obj2=passenger, loc=L].
  3. decrease[obj2=passenger[mod=bus]], loc=L] ⇒ abolish[bus, loc=L].
  4. enforce[obj=two-way-lane, loc=L] ⇒ dangerous[obj=pedestrian, loc=L].
  5. enforce[obj=two-way-lane, loc=L], turn-on[act=bus, obj=lights, loc=L] ⇒ dangerous[obj=pedestrian, loc=L, pol=0].
  6. enforce[obj=two-way-lane, loc=L] ⇒ dangerous[obj=enteringcar, loc=L].
  7. enforce[obj=two-way-lane, loc=L], set-up[obj=road-sign, loc=L] ⇒ dangerous[obj=entering-car, loc=L, pol=0].
- Facts
  1. enforce[obj=one-way-system, loc=Midosuji]
  2. enforce[obj=two-way-lane, loc=Midosuji, pol=0].
  3. change[obj2=bus-route, loc=Midosuji].
  4. change[obj2=passenger[mod=bus]], loc=Midosuji].
  5. enforce[obj=two-way-lane, loc=London].
  6. dangerous[obj=pedestrian, loc=London, pol=0].
  7. turn-on[act=bus, obj=lights, loc=London].
- Judgments
  1. ng[obj=abolish[mod=bus]].
  2. ng[obj=dangerous[obj=_]].

Figure 1: Contents of the Beliefs

of affairs to the system as the argument goals. In the table, A means some state of affairs, and $\bar{A}$ means the negative state of affairs of A. If a judgment $g(A)$ exists in the system's belief, then the system believes A to be good.

Argument goal	Assertion	Correspondent Judgement
$must(A)$	It must be A	$ng(\bar{A})$
$hb(A)$	It had better be A	$g(A)$
$may(A)$	It may be A	$\neg ng(A)$

Dulcinea converts the given argument goal to the correspondent judgment, and shows that this judgment is supported by its beliefs.

2.5 Constituents of the Argument Text

The argument texts consist of the grounds for the argument goal, the expected opposing arguments, its refutation and the examples. The plans are prepared for each constituent to generate its content. A brief description of the constituent follows.

- **Argument goal**

  The argument goal is given to the system first. It provides the system's standpoint. It is the conclusion of the text, which is mostly placed at the end of the text.

  Ex. 1 .... *Therefore, the two-way lane* [1] *must be enforced.*

- **Ground**

  The grounds are necessary to justify the argument goal. The type of argument goal and the beliefs are used to select the proper plan for generating the grounds. The plans that are very restricted based on the reasons for the states of affairs will be described in detail in Section 3. An example of the grounds is given below.

  Ex. 2 *Because the bus service stops if a two-way lane is not enforced, ...*

- **Opposing argument and its refutation**

  Showing the grounds is enough to justify the argument goal. But, add to this any expected opposing argument and its refutation increases the persuasiveness of the text. The system adds the pseudo-ground of the opposite argument goal as the opposing argument, and points out that it is incorrect by refuting it.

  Ex. 3 *Indeed enforcing the two-way lane seems to be dangerous for pedestrians, but they are safe if the buses turn their lights on.*

- **Example**

  The argument text with the example is more persuasive.

  Ex. 4 *For example, the number of passengers using the bus decreased, because the two-way lane was not enforced.*

2.6 Argument Graph

The semantic contents that consist of the above parts are represented by the argument graph. Figure 2 is an example of the argument graphs, which insist the argument goal *"The two-way lane must be enforced"*.

Each node in the graph represents a state of affairs. `Ng` in the nodes indicates that the state of affairs is regarded as *no_good*, and `af` indicates that the system assumes that this is true. Nodes (2)~(5) with the assumed node (1) represent the grounds for justifying of the argument goal. The `cause` link in the graph means a general causation, and the `p` link is used to represent the assumed node. The term in the node (2) `enforce(two-way-lane;0)` is the negative state of affairs of `enforce(two-way-lane)`. The system regards node (5) as *no_good* and node (2), the negative state of affairs of the argument goal which causes the *no_good* state of affairs, as node (5). Therefore, this causal relation is the ground for the argument goal. The `anti` link means the linked graph has contents opposite to the ground, and the `deny` link shows that the node seems to be caused, but is denied by the linked graph. The details of the ground, the opposing argument and the refutation of it are given in Section 3.

[1] A two-way lane is a lane which allows buses to drive the wrong way up a one-way street.

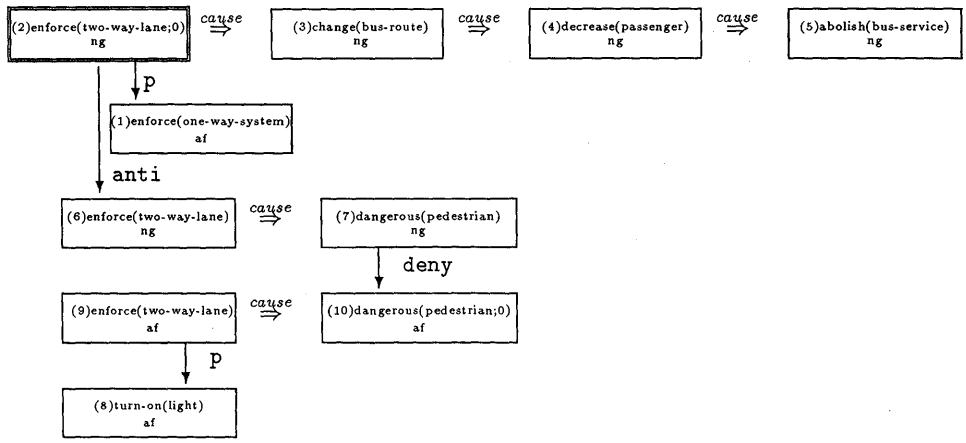


Figure 2: Argument Graph

2.7 A Structural Gap between the Argument Graph and a Linguistic Text Structure

The system realizes the semantic contents by natural language expression. However, the semantic text structure does not always correspond to the linguistic text structure. The various relations in the argument graph such as causation, temporal sequence, condition and assumption are expressed by various natural language connective expressions by considering the system's standpoint and beliefs. The direct realization of the propositional content makes for unnatural text. Therefore, the system must consider the judgment in relation to the propositional content and the role of the block that the propositional content is placed on such as the grounds and the opposing argument. One of the correspondent natural language expressions to the argument graph in Figure 2 is given below.

Ex. 5 *If the one-way system is introduced into a street*$_{(1)}$*, and a two-way lane is not enforced in the street*$_{(2)}$ *then the route of the bus service changes*$_{(3)}$*, the number of passengers decreases*$_{(4)}$*, and, unfortunately, the bus service eventually stops* $_{(5)}$.

Indeed enforcing the two-way lane$_{(6)}$ *seems to be dangerous for pedestrians*$_{(7)}$*, but they are safe*$_{(10)}$ *if the buses turn their lights on*$_{(8)}$*, even if the two-way lane is enforced*$_{(9)}$.

Nodes (2)$\sim$(5) are linked by the cause link, but the surface expressions connect them naturally in a variety of ways. In (5), *"unfortunately"* is used to express the writer's negative attitude, this word communicates efficiently that nodes (2)$\sim$(5) are the grounds. The expression *"Indeed* $\sim$ *seems to be* $\sim$*"* and *"even if* $\sim$*"* are used, because (6)$\sim$(7) represent content that opposes the argument goal. These are denied by (8)$\sim$(10). Since (6)$\sim$(10) have a different content from (1)$\sim$(5), two separate paragraphs are formed.

All these things make it clear that the surface natural language expression reflects the system's standpoint and beliefs besides the propositional contents. However, since there is a gap between the semantic contents and the natural language expression, the direct realization from the semantic data structure needs complicated processing because their is too much information to be referred to and a large variety of decision orders. To realize the proper expressions, as in the example above, it is necessary to not only refer to the relations between each state of affairs, but to consider how the partial structure relates to the whole text. A limited natural language expression is likely to be realized to avoid complexity, and such expressions cannot affect the reader.

In the early stages of the generation process, the organization of the linguistic text structure based on the linguistic strategy for the argument is important to realize the persuasive text, which utilizes the rich expressiveness of natural language. The linguistic strategy consists of the prescriptive descriptions of the developing topics and the local plans for each constituent to represent the local relations. In addition, we need the abstract representation form to represent the text structure generated as a result of structure planning.

2.8 FTS (Functional Text Structure)

We introduced the FTS as the abstract representation form. The FTS is able to represent information such as the writer's judgments, necessary to generate coherent text, and to reflect the writer's standpoint besides the propositional contents. Both the local relations between the states of affairs and the global construction of the text are described together.

FTS is a text structure representation form which represents the functional relations that hold within a piece of text. FTS consists of the FTS-term, order constraints and gravitational constraints. The order constraints and the gravitational constraints are optional.

FTS-term: The data structure that represents functional dependencies that hold within a piece of text. FTS does not fix the order of the sentences.

Order constraint: The constraint of the order between two sentences. The order constraint S1<S2 means that the text in which sentence S2 comes after sentence S1 is preferable.

Table 1: Attribute Labels in the FTS-term

Labels	Description of attribute
thesis	A conclusion of the FTS-term
reason	A reason of the thesis
anti_t	An opposing content of the thesis
crecog	A cause of the thesis
exampl	An example of the thesis

Gravitational constraint: The constraint of the distance between two sentences. The gravitational constraint S1–S2 means that the text in which sentence S1 is near sentence S2 is preferable.

Table 1 is a list of attributes to describe the FTS-term. These attributes take the FTS-term recursively as its value except `thesis` that takes a belief content as its value.

FTS produces various surface text structures by deciding the order of the sentences, and whether to connect two adjacent sentences or not, as well as the type of the connectives. The order of the sentences and the connection of the adjacent sentences is important to make the text comprehensible. Our system is able to generate coherent text in regards to sentence order and connection by selecting the best surface text structure from the structures that FTS can generate. The selection is based on criteria we will describe later.

3 Overview of the System

The argument text generation system Dulcinea consists of the four modules described below.

- **Generation of semantic contents of arguments**

  This module creates a data structure called an Argument Graph which represents the semantic content of arguments to justify a given argument goal according to the system's own beliefs.

- **Linguistic organization with argument strategy**

  This module creates an FTS from a given argument graph using linguistic knowledge. The FTS represents a whole text structure.

- **Clause level organization of orders and connections**

  Each leaf node in the FTS corresponds to a clause in natural language. This module adds order and connection information from each clause to the FTS.

- **Realization of texts**

  To realize natural language text from the FTS, appropriate words are selected. Tense, aspect and mood are fixed in this module.

These four modules are connected in sequence (Figure 3). Details on each module are described in the est of this section.

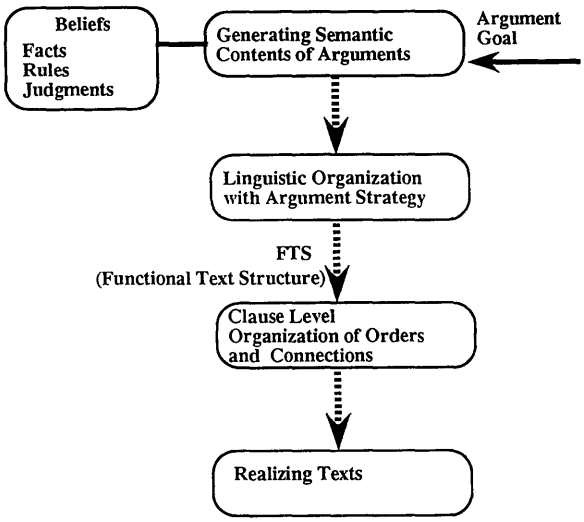


Figure 3: Dulcinea's Architecture

3.1 Generating Semantic Contents of Arguments

This module creates an argument graph which represents the semantic content of the argument from the given goal. The module refers to a knowledge base which contains the system's own beliefs while creating the argument graph.

As described in the previous section, an argument consists of three different parts: grounds for the argument, refutations of the opposing arguments, and examples of the arguments or refutations. Every argument has at least one ground argument, whereas refutations of the opposing argument and examples are optional to the argument. We will describe the procedure for creating each part and combining those parts into one argument.

1. **Generation of grounds**

   The procedure for creating ground differs according to the type of goal. The procedures are as follows.

   (a) **goal type 1** (*must*, *hb*)

   The module searches a reason for believing a judgment corresponding to a given goal. If there is a rule in beliefs which predicts a result state B from a state A, and state B is believed to be good ($g(B)$), then state A is also believed to be good ($g(A)$).

$A_1, A_2, ..., A_n \Rightarrow B$ $A_2 \sim A_n$ $g(B)$ $g(A_1)$	$A_1, A_2, ..., A_n \Rightarrow B$ $A_2 \sim A_n$ $ng(B)$ $ng(A_1)$

   In the course of applying these schemas, states $A_2 \sim A_n$ are proved by applying rules backward. If those states cannot be proved by the schemas below, the module assumes those states hold in its belief.

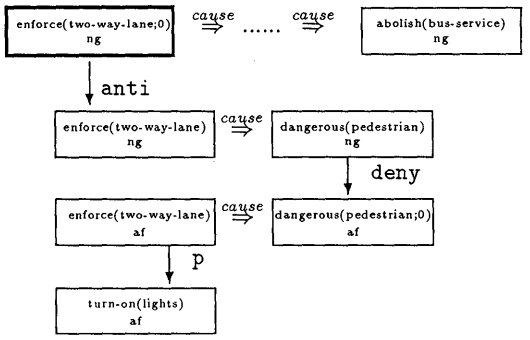


Figure 5: Generation of Refutation of Opposing Argument

$$\frac{A_1, A_2, ..., A_n \Rightarrow B \quad A_1 \sim A_n}{B}$$

The result of application of rules is represented in an argument graph. Figure 4 shows an example.

(b) **goal type 2** (*may*)

A goal of the form $may(A)$ corresponds to a judgment $\neg ng(A)$. We cannot obtain this type of judgment using the schema above. we define the semantics of $\neg ng(A)$ as follows, "There seems to be grounds for a judgment $\neg ng(A)$. But, in fact, there is a refutation to the argument"

This idea is also used in the creation of refutations of an opposing argument.

2. **Generation of opposing arguments and their refutation**

An argument A_1 whose goal is contrary to the goal of argument A_2 is called an opposing argument of A_2. Its goal and its opposing argument's goal are listed below.

Argument goal	Opposing goal
$must(A)$ $(= ng(A))$	$ng(A), g(A)$
$hb(A)$ $(= g(A))$	$ng(A), g(A)$

The module creates the pseudo-ground for the goal opposing the original goal. Find, then, creates the refutation to the opposing argument. Figure 5 shows an example of the refutation of the opposing argument.

3. **Generation of examples**

We define a pair of facts which are unifiable to a rule as an "example" of the rule. By attaching examples to the rules in an argument, we can reinforce the argument (See Figure 6).

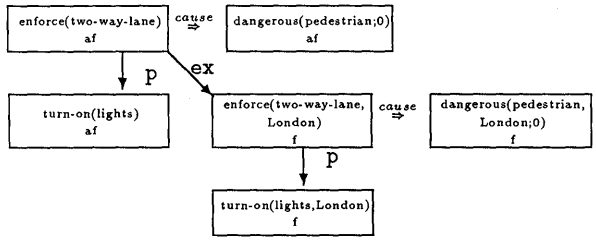


Figure 6: Generation of the Example

3.2 Linguistic Organization with Argument Strategy

This process applies some linguistic argument strategies to an argument graph and constructs an FTS of an argument text. Since the argument graph expresses only the semantic content of the argument, the structure of the graph is independent of the natural language expressions to be generated. Therefore, in order to generate the argument text, the argument graph should be translated into the FTS, which can be transformed into suitable natural language expressions.

First of all, basic constituents in the argument graph such as the ground, the example and the refutation of the opposing argument are recognized. Then the order of these constituents is decided according to the prescriptive knowledge. The order is described using the order constraints of the FTS.

For instance, the opposing arguments are placed before the argument goal. The examples are placed after the ground. The ground is realized earlier than its opposing arguments and refutations of it.

At the same time, each constituent is transformed into the FTS-term according to the transformation rules. Those constituents which cannot be used for the argument or would make the text unnatural are ignored.

The following shows the transformation rules defined for each constituent of the argument graph.

1. **Generation of the ground**

A causal relation in the argument graph is transformed into a new term which has a pair of labels `cause` and `result`. If the causal relation has precondition link `p`, then the content of the precondition with a label `p_cond` is added to the term. For example, the argument graph in Figure 4 is transformed into the following FTS-term. The FTS-term which represents the ground for the given argument goal has an attribute `fts_type` and its value `main`.

```
[thesis=[set={
        [thesis=[p_cond=enforce(two-way-lane),
                 cause= enforce(two-way-lane;0),
                 result=change(bus-route)]],
        [thesis=[cause= change(bus-route),
                 result=decrease(passenger)]],
        [thesis=[cause= decrease(passenger),
                 result=abolish(bus-service)]]}],
 fts_type= main]
```

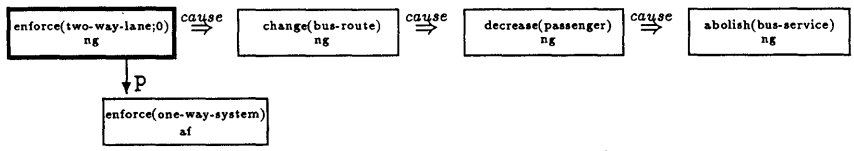


Figure 4: Generation of the Ground

2. **Generation of the opposing argument and the refutation of it**

   The FTS-term which represents the opposing argument is generated with a label `anti_t` in the FTS-term which represents the ground. The contents of the opposing argument in the argument graph are transformed into the FTS-term in the same way as the transformation of the ground part.

   The FTS-term which expresses refutation of the opposing argument has an attribute `fts_type` with the value `anti_deny`. The following shows the FTS-term corresponding to the argument graph in Figure 5.

```
[set={
 [thesis=t1:...,
  fts_type= main],
 [thesis=t2:[though=enforce(two-way-lane),
             assume=turn-on(lights),
             result=dangerous(pedestrian;0)],
  anti_t=t3:[thesis=
             [seem=
              [thesis=
               [cause=enforce(two-way-lane),
                result=dangerous(pedestrian)]]]],
  fts_type=anti_deny]
    }]
order constraints: t1<t2, t3<t1
```

3. **Generation of the example**

   The FTS-term which stands for the example of the ground is generated with the label `exampl` in the FTS-term expressing the ground. The FTS-term that corresponds to the argument graph in Figure 6 is as follows.

```
[thesis=
  [though=enforce(two-way-lane),
   assume=turn-on(lights),
   result=dangerous(pedestrian;0)],
 exampl=
  [thesis=dangerous(pedestrian,London;0),
   crecog=[set={
           [thesis=enforce(two-way-lane,London)],
           [thesis=turn-on(lights,London)]}]]]
```

3.3 Clause Level Organization of Orders and Connections

This module defines the connection relation of each sentence in a given FTS, and generates the surface structure of a whole text. In general, a number of surface structures can be generated from one FTS. In order to generate one plausible surface structure, the module processes the FTS in two steps.

Table 2: Criteria for Connection Relation

Depth of a memory stack	The depth of a memory stack should be shorter.
Number of bad dependency structures	The number of bad dependency structures should be smaller in a text.
Structural similarity	The structure of the surface text should be similar to the FTS.
Number of connectives with negative statements	Not more than two connectives to introduce negative statement should appear in a sentence.
Number of connecting two clauses	Two clauses should not be connected more than a certain number of times.
Gravitational constraint	Two sentences under the gravitational constraint should be placed close.
Stability of topics	Sentences should be ordered so as not to change the topic frequently.
Connecting two implications	Two implications A→B and B→C should be realized in this order
Sentence order similarity between the ground and the example	The sentence order of the example should be similar to the ground.

1. Generates every possible connection relation from the given FTS.

2. Evaluates those connection relations based on the criteria in Table 2, and chooses the best connection relation.

Using the criteria for connection relation in Table 2, the module adds an `order` attribute which represents sentence order and a `conn` attribute which represents the connection of two sentences to the FTS. The surface expression for connectives are specified by the type of the connections (Table 3).

We illustrate the criterion for the bad dependency structure using the FTS below.

```
[thesis= t1
 anti_t= t2
 exampl= t3]
t2 < t1,  t2 < t3
```

In this case, we can generate sentences in two different orders.

1. `t2<t1<t3`
2. `t2<t3<t1`

Figure 7 represents the dependency structure of each text. Since dependency structure 2 does not have direct dependency between `t2` and `t3`, the reader cannot find the semantic dependency of `t3` when `t3` is reached while reading this text. They can find the semantic dependency only after reading through `t1`. This means that the text in order 2 is much more difficult to understand

Table 3: Type of Connections and Their Expressions

deduction	s	shitagatte,dakara,yotte,yueni,...
	c	~kara,~node,...
causation	s	sonokekka,sonotame,...
	c	~tame,~(ren'youkei),...
reason	s	nazenara~karadearu,toiunoha~karadearu,...
	c	×
development	s	suruto, ...
	c	~to,~(ren'youkei),...
negation1	s	shikashi,daga,...
	c	~ga,...
negation2	s	~ga,...
	c	×
juxtaposition	s	mata,...
	c	~shi,...
example	s	tatoeba,jissai,...
	c	×
generalization	s	konoyouni,...
	c	×
presentation	s	(just placed continuously)
	c	~ga,
implication	s	×
	c	naraba,reba,to,...
addition	s	×
	c	te,(ren'youkei),...
concession	s	×
	c	temo,tatoe~temo,...

s:separate
c:connect

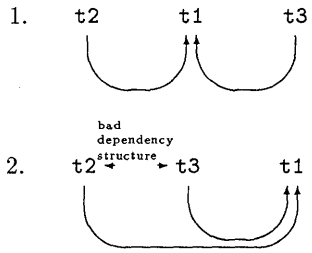


Figure 7: Dependency Structure

than that in 1. We call the dependency structure in 2 the bad dependency structure.

In evaluation of the bad dependency structure, order 2 is preferred to 1, and the attributes' values become the following.

```
[thesis= t1
 anti_t= t2
 exampl= t3
 order= [2,3,1],
 conn= [(negation1,s),(example,s)]]
```

3.4 Realizing Texts

This process realizes the content of the FTS added `order` and `conn` attributes in terms of natural language expressions. The FTS with `order` and `conn` is the tree structure which represents the syntactic structure of the whole text. The leaves of the tree structure are realized as clauses. Syntactic structural relations which hold at a higher level than clauses, such as the relation between clauses and the relation between sentences, have already been generated by adding `order` and `conn` to the FTS.

For each term in the tree structure, lexicons correspond to each object in the term. Here, suffixes expressing the functions for each object, tense and aspect expressions of predicates and the system's judgment expressions are all decided. Then, connectives which represent the relations between terms are determined by Table 3.

Among the causal relations between terms, those that have been described as a rule in the beliefs are represented as an "implication", which is a strongly dependent connective relation. The following rule in the belief

change(bus-route)⇒decrease(passenger)

will be realized *"If the bus route is changed, the passengers decrease."* The relation between `thesis` and `reason` described in the FTS-term is represented as a "deduction", which is a weakly dependent connective relation. For instance, the FTS-term:

```
[thesis=must(enforce(two-way-lane)),
reason=abolish(bus-service)]
```

will be expressed *"The bus service will be abolished. Therefore, a two way lane should be enforced."*

4 Example

In this section, we show an example of Dulcinea's argument. The beliefs used for the argument are shown in Figure 1.

When the argument goal

```
must(cont=enforce(obj=two-way-lane)),
```

which means *"The two-way lane must be enforced."*, is given to Dulcinea, it generates the argument graph shown in Figure 9 according to the beliefs.

Then this argument graph is transformed into the FTS, to which information on sentence order and connectives are added afterwards. The FTS with these two kinds of information is shown in Figure 10.

From this FTS structure the argument text shown in Figure 8 is realized.

5 Conclusion

We have described the argument text generation system Dulcinea, which generates text to justify the argument goal. The text, which reflects the standpoint and the judgment of the system, is represented by the FTS. The FTS represents not only the semantic contents of the argument but the system's standpoint, the judgments and the linguistic constraints.

In addition to the generation frame-work, we have investigated the argument strategy to generate coherent and persuasive argument texts. This strategy is plausible for the multi-paragraph text generation in a very narrow domain, but we believe that it is one of the answers to the question: "What relations, plans and schemas are necessary to support the planning of coherent multi-paragraph texts?".

御堂筋で一方通行を実施して，逆行レーンを実施しなかった．その結果，バスルートが変化した．そのため，バスの乗客が40%減少してしまった．このように，一方通行を実施するときに，逆行レーンを実施しなければ，バスルートが変化し，バスの乗客が減少する．また，バスが廃止されてしまう．

一方，逆行レーンを実施すれば，歩行者が危険なようにみえる．しかし，バスのライトを点灯すれば，逆行レーンを実施しても，歩行者は危険でない．したがって，逆行レーンを実施しなければならない．

Generated Text

When the one-way system was introduced in *Midosuji* street, a two-way lane was not enforced. As a result, the route of the bus service changed. Therefore, the number of passengers decreased by 40%. In this way, when a one-way system is introduced to a street, if a two-way lane is not enforced, then the route of the bus service changes, and this makes the number of passengers decrease. Finally, the bus service is abolished.

On the other hand, enforcing the two-way lane seems to be dangerous for pedestrians. But they are safe if the buses turn their lights on. Therefore, the two-way lane must be enforced.

Translated from Japanese

Figure 8: Argument Text

To generate more coherent and natural texts using the rich expressiveness of natural language, some user model and more complicate conversational settings will be necessary in the future.

Acknowledgments

Thanks are due to TANAKA Yuichi for reading the draft and making a number of helpful suggestions. We also wish to thank all the members of the Sixth Research Laboratory and the members of working group NLU for valuable advice.

References

[Appelt 1988] Douglas E. Appelt. Planning natural-language referring expressions. In David D. McDonald and Leonard Bolc, editors, *Natural Language Generation Systems*. Springer-Verlag, 1988.

[Danlos 1984] Laurence Danlos. Conceptual and linguistic decisions in generation. In *the Proceedings of the International Conference on Computational Linguistics*, 1984.

[Hovy 1985] Eduard H. Hovy. Integrating text planning and production in generation. In *the Proceedings of the International Joint Conference on Artificial Intelligence*.

[Hovy 1987] Eduard H. Hovy. Interpretation in generation. In *the Proceedings of 6th AAAI Conference*.

[Hovy 1988] Eduard H. Hovy. *Generating Natural Language under Pragmatic Constraints*. Lawrence Erlbaum Associates, Publishers, 1988.

[Hovy 1990a] Eduard H. Hovy. Pragmatics and natural language generation. *Artificial Intelligence*, 43:153–197, 1990.

[Hovy 1990b] Eduard H. Hovy. Unresolved issues in paragraph planning. In *Current Research in Natural Language Generation*. Academic Press, 1990.

[Joshi 1987] Aravind K. Joshi. Word-order variation in natural language generation. In *the Proceedings of 6th AAAI Conference*.

[Mann and Thompson 1987] W. C. Mann and S. A. Thompson. Rhetorical structure theory: Description and construction of text structures. In *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics*. Dordrecht: Martinus Nijhoff Publishers, 1987.

[McDonald and Pustejovsky 1985] David D. McDonald and James D. Pustejovsky. Description-directed natural language generation. In *the Proceedings of the International Joint Conference on Artificial Intelligence*.

[McKeown 1985] K. R. McKeown. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text*. Cambridge University Press, 1985.

[McKeown and Swartout 1988] K. R. McKeown and W. R. Swartout. Language generation and explanation. In Michael Zock and Gérard Sabah, editors, *Advances in Natural Language Generation*, volume 1. Ablex Publishing Corporation, 1988.

[Metter 1990] Marie W. Meteer. The 'generation gap' the problem of expressibility in text planning. Technical report, BBN Systems and Technologies Corporation, 1990.

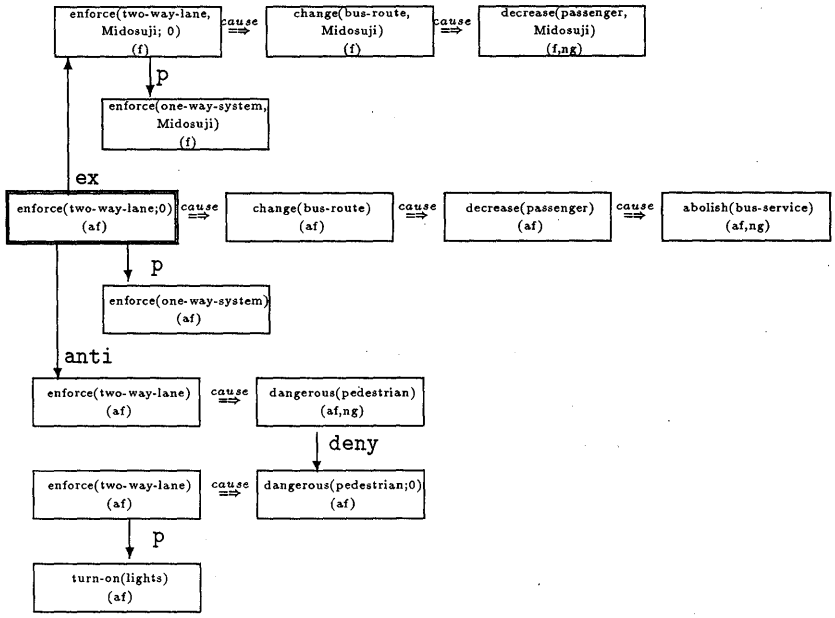


Figure 9: The Argument Graph

```
[thesis= 1:must[cont= enforce[obj=two-way-lane]],
 reason= 2:[set={
    1:[thesis= [set={
          1:[thesis= 1:[set={
               1:[thesis= [p_cond= 1:(enforce[obj=one-way-system],af),
                           cause= 2:(enforce[obj=two-way-lane,pol=0],af),
                           result= 3:(change[obj2=bus-route],af),
                           order= [1,2,3], conn= [(condition,c),(implication,c)]]],
               2:[thesis= [cause= 1:(change[obj2=bus-route],af),
                           result= 2:(decrease[obj2=passenger[mod=bus]],af),
                           order= [1,2], conn= [(implication,c)]]]},
               order= [1,2], conn= [(development,c)]],
            exampl= 2:[thesis= 1:(decrease[obj2=passenger[mod=bus],amo=40%,ten=prec,loc=Midosuji],f,ng),
                       crecog= 2:[thesis= 1:(change[obj2=bus-route,loc=Midosuji],f),
                                  crecog= 2:[set= {
                                     1:[thesis= (enforce[obj=one-way-system,loc=Midosuji],f) ],
                                     2:[thesis= (enforce[obj=two-way-lane,loc=Midosuji,pol=0],f)]},
                                     order= [1,2], conn= [(juxtaposition,c)]],
                      order= [2,1], conn= [(causation,s)]],
            attent= [{loc,Midosuji}],
            order= [2,1], conn= [(causation,s)]]]},
     order=[2,1], conn=[(generalization,s)]],
          2:[thesis= [cause= 1:(decrease[obj2=passenger[mod=bus]],af),
                      result= 2:(abolish[obj=bus],af,ng),
                      order= [1,2], conn=[(implication,c)]],
            order= [1,2], conn= [(juxtaposition,s)]],
     ftst_type= main],
   2:[thesis= 1:[though= 1:(enforced[obj=two-way-lane],af),
                 assume= 2:(turn-on[obj=lights[mod=bus]],af),
                 result= 3:(dangerous[obj2=pedestrian,pol=0],af),
                 order= [2,1,3], conn= [(implication,c),(concession,c)]],
      ftst_type= anti_deny,
      attent= [{obj2,pedestrian}],
      anti_t= 2:[thesis= [seem= [cause= (enforce[obj=two-way-lane],af),
                                 result= (dangerous[obj2=pedestrian],af,ng),
                                 order= [1,2], conn= [(implication,c)]]
                                 order= [1], conn= []]],
      order= [2,1], conn= [(negation1,s)]] },
   order= [1,2], conn= [(change,s)]],
order= [2,1], conn= [(deduction,s)]]
```

Figure 10: FTS with order and conn attributes

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Situated Inference of Temporal Information

Satoshi Tojo Hideki Yasukawa†

Mitsubishi Research Institute, Inc.
8-1, Shimomeguro 1, Meguro-ku, Tokyo 153, JAPAN
(phone) +81-3-3536-5813 (e-mail) tojo@mri.co.jp

Institute of New Generation Computer Technology (ICOT)†
Mita Kokusai Bldg. 21F
4-28 Mita 1, Minato-ku, Tokyo 108, JAPAN
(phone) +81-3-3456-3194 (e-mail) yasukawa@icot.or.jp

Abstract

Representations of natural language, describing the same state of affairs may differ between speakers because of their different viewpoints. In this paper we propose the concept of perspectives that are applied to situations, to explain this variety of the representation of an infon, with regard to time. We define the perspective by the relation theory of meaning, namely the relative locations in mind between the described situation, the utterance situation, and the infon. Our aim is to model a situated inference system that infers temporal features of a sentence from the partial temporal information each lexical item carries. For this purpose, we apply our notion of perspectives to actual tense and aspects that are used as lexical temporal features, and in addition we inspect the validity of our formalization for verbs. We show the inference system in the logic programming paradigm, and introduce an ambiguity solver for Japanese *-teiru* that may have multiple meanings, as an experiment of our framework.

1 Introduction

The most common way to represent time is to assume that it is a one dimensional line which extends both to the eternal past and to the eternal future, and a point called 'now' which moves along the line at a fixed speed. The semantics of time in natural languages, so often associated with this physical time parameter 't', has been dealt with. However, the introduction of parameter 't' seems too strong for natural languages, contrary to the case of physical equations. Actually, we cannot always map the temporal property of verbs or temporal anaphora on the time axis correctly.

In opposition to this view of the time, we have been forced to loosen the strongest topology of physical time in some ways. One of the most famous works to represent so-called 'coarse' time is the interval-based theory by Allen [Allen84]. Kamp [Kamp79] proposed event calculus where he claimed that 'an instant' was relatively defined by all the known events in his DRT. Our work is to extend this temporal relativity. We will mainly pay attention to the temporal structures of tense, aspects, and verbs within the framework.

From the viewpoint of the history of situation theory, the notion of a spatio-temporal location, or simply a location, was proposed to represent the four-dimensional concept of time and place. In the early stages of situation theory, situations and spatio-temporal locations were distinguished [Barwise83] as: *In s, at l σ holds.* However the consideration of spatio-temporal location seems to have been rather neglected since then, and we have only found Cooper's work [Cooper85] [Cooper86] to give a significant interpretation of locations for time semantics.

The authors have worked for the formalization of a temporal location as a meaning carrier of temporal information [Tojo90]. In this paper, we aim to show a

paradigm of an inference system that merges temporal information carried by each lexical item and resolves any temporal ambiguity that a word may have. First we review the role of the temporal location following Cooper's work. Our position is to regard temporal locations of infons and situations as mental locations. We define a temporal perspective toward a situation, which decides how an infon is verbalized, in terms of relative locations of situations and infons. In the following section, we will give accounts for several important temporal features of tense and aspects by the perspectives, not only to define the basic information for the intended situated inference but also to see the validity of our formalization. In the following section, we discuss the computation system that infers temporal features of a natural language sentence. We have implemented an experimentation system of the ambiguity solver in Japanese *-teiru* with a knowledge representation language $\mathcal{QUIXOTE}$, developed at ICOT (Institute of New Generation Computer Technology, Japan).

2 Situations with perspectives

The temporal information in our mind seems preserved in a quite abstract way, and temporal span or duration are relative to events in the mind. In this section, we will discuss the structure of those subjective views for time, and for the real situation.

2.1 Real situations and perspectives

We often write down an infon in the following way:

$$\ll relation, parameters \gg$$

However none have been concerned with those *labels* for the *relation* in an infon. For example, should we admit such *relations* as those contains tense and aspect? If so, and if the following supporting relations are valid:

$$s \models \ll swim, john \gg$$
$$s' \models \ll swam, john \gg$$
$$s'' \models \ll is\text{-}swimming, john \gg$$

then how should we describe the relation in s,s', and s''? Are we making different expressions for the same real situation?

We hypothesize a virtual physical world, or in other words an ontological world which was originally proposed as a real situation [Barwise83]. According to the notion of real situations, we can assume that there is proto-lexicon in the world though there are many different ways to verbalize them. We may call those infons that are not yet verbalized *proto-infons*.[1] We can regard proto-infons as the genotype of infons; to describe a proto-infon to make the phenotype one is to give *rel* and *roles* in natural language with a certain viewpoint. We propose an idea of perspective which gives this notion of view next.

In the scheme of individuation, Barwise regarded all linguistic labels as being encoded in the situation itself already [Barwise89]. We will not discuss the adequacy of this idea in this paper, however it gives us a way to formalize situations with perpsectives as follows. In order to state a formula of the form $s \models \sigma$, we are required to assume a certain observer who has cut out s as a part of the world and has paid attention to information σ, so that the formula must already contain someone's view or perspective. In that meaning, in S or in σ, the basic lexicon must be included as linguistic labels. For example, if the observer is a Japanese, Japanese language labels should be used to describe the information. From this point of view, we assume that in the formula of the support relation between a situation and an infon someone's perspective already exist.

$$s \models \sigma \Leftrightarrow \mathcal{P}(s' \models \sigma')$$

It is an open question whether we can strip off all the external perspectives from a support relation, as below:

$$\mathcal{P}_1(\mathcal{P}_2(\cdots \mathcal{P}_n(s \models \sigma)\cdots))$$

Even if we can, this must not be the only way to choose a sequence of $\mathcal{P}_i$'s.

2.2 Temporal perspective

We concern ourselves with the temporal part of the spatio-temporal location of the situation theory here. The natural way to do this is to assume that there is a support relation:

$$s \models \ll \cdots \gg$$

that is already verbalized even though the *relation* inside the infon do not have tense nor aspects. Our formalization is as follows. There is a *perspective* $\mathcal{P}$ for a support relation that adds tense and aspect.

[1] This is a reinterpretation of the concept of *information* in the real situation.

$$\mathcal{P}(s \models \ll rel, \cdots \gg)$$
$$\Downarrow$$
$$\mathcal{P}(s) \models_t \ll rel\_with\_tense\_aspect, \cdots \gg$$

Here, we assumed that the perspective is decomposable to both sides of the supporting relation, the meaning of which is assumed to be independent of each perspective though we add the subscript 't' to represent if $\mathcal{P}$ is temporal. We may omit the subscript hereafter to avoid confusion.

The next work we need to do is to define the structure of a $\mathcal{P}$.

2.3 Relation theory of meaning with regard to time

Suppose that the mental descripton of the temporal length, or the duration, of an infon can be written as $\|\sigma\|_t$. In the same way, we can assume the temporal size of a situation such as $\|s\|_t$, if we use the situation as some time-space expansion. In this case, which is the temporal location l_t, $\|s\|_t$ or $\|\sigma\|_t$? And also how should we interpret the supporting relation with regard to time? One possibility is the inclusion of intervals:

$$s \models \sigma \quad \Leftrightarrow \quad \|s\|_t \supseteq \|\sigma\|_t$$

while other people may say

$$s \models \sigma \quad \Leftrightarrow \quad \|s\|_t \subseteq \|\sigma\|_t$$

is more felicitous. Actually the authors consider that the plausibility between '$\supseteq$' and '$\subseteq$' depends on the temporal feature of the *rel*-ation of an infon[2].

Although we cannot fix the size and the location of information in the real time scale, we assume we can map them in a relative way with other events inside our minds. In this paper, we do not use the notion of temporal locations l_t on the physical time axis. Instead, we will consider mentally decscribed $\|s\|_t$ and $\|\sigma\|_t$. s was a part of the world cut off by a perspective of a certain observer. In this meaning, we may say that $\|s\|_t$ is the temporal area the observer is paying attention to so that we may name it as the *field of view* of the perspective.

> **Field of view:** To which time part of the event the observer pays attention, that is a mental location of the described situation $\|s\|_t$.

We call $\|\sigma\|_t$ an *in-progress state* [Parsons 90] of σ.

> **In-progress state:** the mental time of the duration of σ, namely from the beginning of σ to the finishing point: $\|\sigma\|_t$.

We need another component for our temporal perspective, that decides tense. Tense should be decided in accordance with the relative position between the described situation and the utterance situation (that offers 'now'), in terms of the 'relation theory of meaning', that is, a natural language sentence 'ϕ' is interpreted as the relation between the utterance situation u and the described situation ϵ, denoted by $u[\![\phi]\!]\epsilon$.[Barwise 83]. According to this theory, we can say that the standpoint of view is the mental location of the utterance situation.

> **Standpoint of view:** From which time point the observer sees the event, that is the mental location of the utterance situation $\|u\|_t$.

We will discuss the temporal features with regard to the above three parameters of $\|\sigma\|_t$, $\|s\|_t$, and $\|u\|_t$[3], as in fig. 1.

We characterize the notion of a perspective as constraints attached to the described situation, each of which is the temporal relation with the utterance situation or with the information as in Fig. 2 (in that figure, the relations are denoted by '$\sim$' or '$\wr$'), where the verbalized information can be identified with the natural language expression.

2.4 Set-theoretical foundation

The real or physical time space has the strongest topology where any two time points can be separated[4] and all the points are totally ordered; in addition, it is a metric space. However our mental recognition for time is much more vague. We regarded that the temporal recognition of one event is the relative position and the relative length of $\|\sigma\|_t$, $\|s\|_t$, and $\|u\|_t$. Therefore, the mental time space we are considering has a very weak topology,

[2] An instantaneous change of state such as '*understand*' seems to require the temporal vicinity ($\|s\|_t \supset \|understand\|_t$) while *stative* verbs should be valid anytime in s ($\|s\|_t \subset \|is\text{-}running\|_t$).

[3] Reichenbach [Dowty 79] distinguished three kinds of temporal information *time of action*, *time of reference*, and *time of speech*. $\|\sigma\|_t$ and $\|u\|_t$ correspond to *time of action* and *time of speech* respectively, and the $\|\sigma\|_t$ is *time of reference*, though we extend the notions as intervals instead of points.

[4] In the meaning of *Axiom of Separation* in topological spaces.

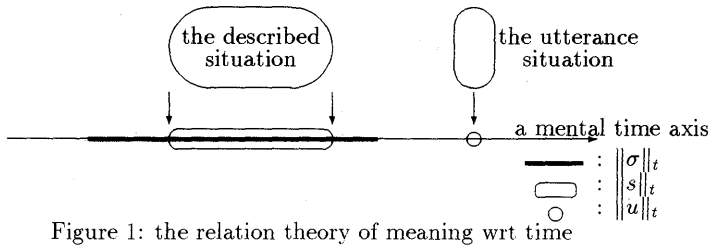


Figure 1: the relation theory of meaning wrt time

$$\begin{array}{ccc} s \models \sigma & & u[\![\phi]\!]s \\ \Downarrow & & \Downarrow \\ \mathcal{P}(s) \models_t \ll \cdots \gg & \equiv & s \left[\begin{array}{c} \sim \ \|u\|_t \\ \wr \\ \sim \ \|\sigma\|_t \end{array}\right] \models_t \phi \end{array}$$

Figure 2: The denotation of the relation theory of meaning

whose open sets are these $\|s\|_t$'s, $\|u\|_t$'s,[5] and $\|\sigma\|_t$'s. We may call them intervals, sets of points, or even points, however they all should be reinterpreted in terms of open sets in the weak topological space.

We will not mention this topological notion hereafter though we may use some set theoretical notations such as '$\subseteq$, $\cup$, $\cap$, ϕ'. We adopt the normal subset relation '$\subseteq$' between $\|\sigma\|_t$ and also $\|s\|_t$ to represent the temporal partiality. In addition, we equip ourselves with the normal temporal order, denoting:

$$T_1 \prec_t T_2.$$

for time intervals T_i's, iff all the time points in T_1 chronologically precede those in T_2. Where an instant may be shared between two sets, we use '$\preceq_t$'. We may drop the subscript 't' hereafter as far as there is no confusion.

3 Tense and aspects as temporal perspectives

We have claimed that the temporal perspective depends upon the individual view of the state of affairs. However, as we use natural languages to communicate with others, we may hopefully receive stereotypical views of things. These stereotypes must be tense and aspects. This implies that we can define tense and aspects by situation types that are independent of each σ. In this section, we will introduce inference rules that infers those stereotypes, that are used as ground rules of the system later.

[5] We can assume that a $\|u\|_t$ is a set that does not contain other sets inside in the topological view, if we were to persist with the interval logic.

3.1 Situated inference rules

We defined the role of a perspective $\mathcal{P}$ as follows: if an infon σ is supported by a situation s and if we add a view $\mathcal{P}$, then σ is transformed to another expression σ'.

$$\mathcal{P}(s) \models \sigma' \ \Leftarrow \ s \models \sigma.$$

or:

$$\frac{s \models \sigma}{\mathcal{P}(s) \models \sigma'}.$$

We defined the notion of perspective by the set theoretical relation between $\|s\|$, $\|u\|$, and $\|\sigma\|$. We denote $s[X]$ for the described situation with a perspective, where the contents of $[X]$ is the temporal consraint that observes the following convention.

1) $s[\prec u], s[\subset \sigma]$, and $s[\supset \sigma]$ are the described situations where $\|s\|_t \prec \|u\|_t$, $\|s\|_t \subset \|\sigma\|_t$, and $\|s\|_t \supset \|\sigma\|_t$ hold, respectively.

2) $s[X, Y]$ means $s[X] \wedge s[Y]$. For example, $s[\subset \sigma, \prec u]$ means s such that $\|s\|_t \subset \|\sigma\|_t$ and $\|s\|_t \prec \|u\|_t$.

3) the relation between σ and u is written as $s[\sigma \prec u]$.

3.2 Tense

We define the tense as the problem of chronological order between $\|s\|_t$ and $\|u\|_t$, independent of $\|\sigma\|_t$.

Let us consider the sentence: 'Ken was running in the park.' There must be a duration of time in which 'Ken runs', which is $\|\sigma\|_t$ where $\sigma = \ll$ run, ken $\gg$ in our definition. However the speaker of the sentence does not necessarily know whether Ken ceased to run at the point this utterance was made. Therefore, whether the verb is past or present does not depend on whether the deed finished. Instead, the only required condition for past is that the area the speaker paid attention to ($= \|s\|_t$) precedes the point at which this utterance was made ($= \|u\|_t$). Fig. 3 depicts this case; in that figure, the deed ($= \|\sigma\|_t$) may or may not finish at the utterance point ($= \|u\|_t$) however both situations can support the same infon: $\ll$ was-running, ken $\gg$.

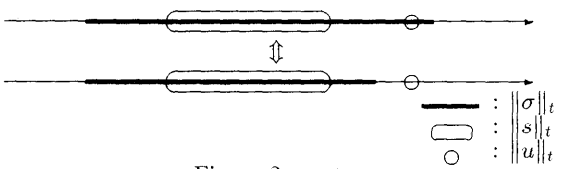


Figure 3: past

The past view for affairs is represented by $\|s\|_t \preceq_t \|u\|_t$. On the contrary, the present tense is represented by $\|u\|_t \subset \|s\|_t$.

We formalize the feature of past and present as follows:

$$s[\preceq u] \models \ll past, \sigma \gg \Leftarrow \; s \models \sigma \tag{1}$$

$$s[\supset u] \models \ll present, \sigma \gg \Leftarrow \; s \models \sigma \tag{2}$$

3.3 Aspects

The study on how we see the temporal features of events has been done in linguistics and we know the variety of aspects. Among the taxonomy, it seems rather proper to pay attention to the following two important features [Comrie 76] though other features may be omitted, because those distinctions can be found in any language. One is:

- the deed is recognized as a duration of time /a point on the time scale of time (durative/non-durative)

and the other is:

- the finishing of the deed is recognized /not recognized (past or perfective/imperfective)

In summary, perfective/imperfective is distinguished, dependent on if the finishing point of $\|\sigma\|_t$ is before $\|u\|_t$. Durative/non-durative depends on if the field of view wraps up $\|\sigma\|_t$ or not.

Durative The most important feature of our method is the distinction of *durative* and *non-durative* aspects by the relation between $\|\sigma\|_t$ and $\|s\|_t$. We interpret the *progressive* feature in terms of our formalization as follows.

The state of progressive is to see the deed as a durative, and seeing a part of the inside of the deed. Namely the speaker does not pay attention to when the deed began, nor to when it will finish. The state is shown in Fig. 4.

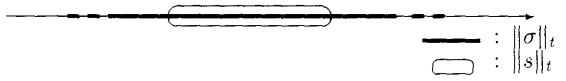


Figure 4: progressive

On the contrary, let us consider the case we do not pay attention to the inside of the deed. When $\|s\|_t$ contains the whole time of the deed, we can conclude that the observer recognized the event as a non-durative one, in which case the event was regarded as a point with no breadth on the mental time axis (Fig. 5), if there is no interaction with other events.[6]

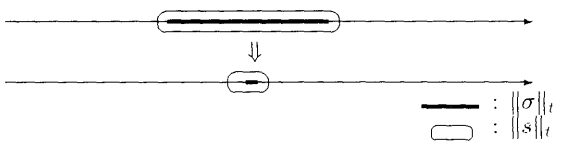


Figure 5: compression to non-durative

A lexical item for the durative view becomes the following:

$$s[\subset \sigma] \models \ll progressive.\sigma \gg \Leftarrow \; s \models \sigma \tag{3}$$

[6] From the topological point of view, a set which does not contain other sets inside nor has intersections with other sets is identified with one of the smallest sets of the space, viz. a point.

Perfect and time of reference As Reichenbach claimed in [Dowty 79], present perfect in English refers to the current state. We have shown that present is represented by the relation that $\|s\|_t$ includes $\|u\|_t$. Therefore, to satisfy this issue, our $\|\sigma\|_t$ must precede the $\|u\|_t$ to represent the *present perfect*. In Fig. 6, we have shown the perspective for the present perfect.

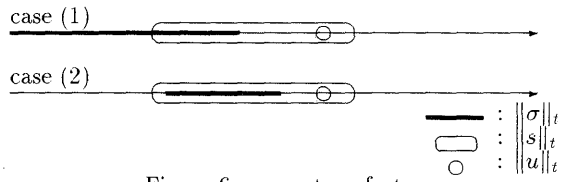


Figure 6: present perfect

In Fig. 6, case (1) shows that perfect is interpreted as a terminative aspect although case (2) shows that perfect is read as an experience.

The perfect view becomes the following:

$$s[\sigma \preceq u] \models \ll perfect, \sigma \gg \Leftarrow \quad s \models \sigma \tag{4}$$

4 Inference of temporal information

4.1 Situated inference

In terms of situated inference, we expect the inference with the following rules:

$$S_0 \models \sigma_0 \quad \Leftarrow \quad S_1 \models \sigma_1, S_2 \models \sigma_2, S_3 \models \sigma_3, \cdots.$$

This sample rule can be interpreted as: if S_1 supports σ_1, S_2 supports σ_2, and so on, then we can infer that S_0 supports σ_0.

This kind of rule can be read as backward chaining from the head, just as with the inference rules of Prolog. We would like to devise a system that computes temporal information, asking questions that corresponds to the head of a rule, and accumulating the temporal information from its body. For example, assume $X, Y, \cdots$ are variables for temporal information.

$$s[X, Y, \cdots] \models \sigma_0 \quad \Leftarrow \quad s[X] \models \sigma_1, s[Y] \models \sigma_2, \cdots.$$

where all the basic, lexical information such as (1), (2), (3), (4), and so on, defined in the previous section, are considered to be the basic rules. The accumulation of temporal information must not be a mere addition; in our case, it must be the merger of different topologies in $\|s\|_t$'s, $\|\sigma\|_t$'s, and $\|u\|_t$'s. The computation, therefore, must be done in the dual mode; one mode is conventional unification and backward chaining, and the other is the merger of $s[X]$ with consistency. This is the reason why we chose the $\mathcal{QUIXOTE}$ language with its concept of modules (situations, in our case) inside of which features can be defined. We will mention the specification later.

We have developed an ambiguity solver for Japanese *-teiru* that can have three different kinds of meaning that depend on the context.

4.2 The problem of Japanese '*-teiru*'

Prior to introducing the ambiguity solver we have developed, we need to give a short tip on Japanese grammar and the problem we tackled.

In the Japanese language, auxiliary verbs are agglutinated at the tail of the syntactic main verb that is the original meaning carrier. In order to compose a progressive sentence, we need to add an auxiliary verb '*-tei-*', and after that we are required to affix a tense marker. We summarize them below for a Japanese verb '*kiru* (to wear)'.

lexical entry	part of speech	meaning
ki-	verb	to wear
-tei-	aux. verb	be -ing*
-ru	affix	present
-ta	affix	past or perfect
ima	adv.	now
zutto	adv.	all the time
san-nen-mae-ni	adv. phrase	3 years ago
ki-tei-ta	verb phrase	was wearing*

The problem lies in the places marked with $*$ in the table above. The meaning marked by $*$ is not the sole meaning of *-tei*: actually we can interpret the auxiliary verb in three different ways, depending on the context.

We show sample sentences below[7].

[7]This example was shown by the members of the JPSG working group in ICOT

ima ki-tei-ru
(be putting on currently)
zutto ki-tei-ru
(wear all the time)
san-nen-mae-ni ki-tei-ru
(have worn three years ago)

We will build our ambiguity solver, focusing on the area of a deed in JPSG framework that corresponds to our $\|s\|_t$. The partial information that each lexical item carries, as defined in the previous section, is utilized, according to the Japanese lexicon table above. Namely we use the inference rules of past (1) /present (2) and perfect (4), for Japanese '*-ru/-ta*'. We use the inference rule of durative (3) for Japanese '*-tei-*'.

The ambiguity of '*-tei-*' is solved as in Fig. 7 where '&' is the merger of information: $X_2 = [\sigma \prec u]$ is incompatible with $[s \subset \sigma]$ in X_5 and $X_6 = [u \subset s]$, so that the value of X_5 necessarily becomes $[\sigma \subset u]$, and this gives '*-tei*' the interpretation of the resultant state.

4.3 Implementation

This section shows an implementation of the treatment of temporal information discussed in this paper. The program is written in the knowledge representation language $\mathcal{QUIXOTE}$[Yasukawa 90],[Yasukawa 92].

4.3.1 $\mathcal{QUIXOTE}$

Terms in $\mathcal{QUIXOTE}$ are extended terms on an order-sorted signature called *object terms*, and written in general as:

$$o[l_1 = o_1, l_2 = o_2, \ldots]$$

where o is an atom called **basic object**, l_1, l_2 are atoms called **labels**, and o_1, o_2 possibly be object terms. The domain of atoms (BO) is ordered and constitutes a lattice $(BO, \preceq, \top, \bot)$.

The *subsumption relation*($\sqsubseteq$) is a binary relation over the domain of object terms, and corresponds to so-called isa-relation. Intuitively, $o_1 \sqsubseteq o_2$ (we say o_2 subsumes o_1) holds if o_1 has more arcs than o_2 and the value of a node of o_2 is larger than the value of the corresponding node of o_1 with respect to $\preceq$-ordering. In $\mathcal{QUIXOTE}$, subsumption constraints can be used to specify an object or the relation among objects.

A rule of $\mathcal{QUIXOTE}$ is a prolog-like clause of the form:

$$m :: \sigma <= m_1 : \tau_1, m_2 : \tau_2, \ldots, m_n : \tau_n \| C.$$

where $m, m_1, \ldots, m_n$ are special extended terms called **module identifiers**, and $\sigma, \tau_1, \ldots, \tau_n$ are extended terms, and C is a set of constraints.

4.3.2 Representation in $\mathcal{QUIXOTE}$

There are several points to be explained, that is, how the notions introduced in the preceding sections are represented.

Object terms are used to represent situations, infons, perspective, and so forth.

First, verbalized infons are represented by object terms of the following form:

$$inf[v\_rel = [rel = R, cls = CLS, per = P], \\ args = Args].$$

The CLS takes the symbols $act_1, act_2, act_3, \ldots$ as its value which indicates the classifications of verbs on what state, that is, in-progress, target, resultant, each verb can introduce. Here's a list of the classification and the states introduced:

$$act_1 \Rightarrow ip, tar, res$$
$$act_2 \Rightarrow ip, res$$
$$act_3 \Rightarrow tar, res.$$

The relationship among these three classes is given by the following subsumption definition.

$$act_1 \sqsupseteq act_2;;$$
$$act_1 \sqsupseteq act_3;;$$

Thus, the verb "*ki-*" can introduce all the three states, while the verbs like "*hashi-*" could not introduce *tar*-state.

For example, the verbalized infon corresponding to the sentence "John is running" is represented as follows:

$$inf[v\_rel = [rel = run, cls = act_1, pers = P], \\ args = [agt = john]].$$

where P is the temporal perspective whose field of view is in-progress and point of view is the *present*. A perspective is also represented by a pair of two object terms as follows:

$$([fov = Fov], [pov = Pov]).$$

where $Fov \in \{ip, tar, res\}$ represents the field of view, and $Pov \in \{pres, past\}$ represents the point of view.

$$\dfrac{\dfrac{s \models \ll \gg}{s[X_2] \models \ll \text{zutto} \gg} \qquad \dfrac{\dfrac{s \models \ll \gg}{s[X_4] \models \ll \text{ki} \gg} \quad \dfrac{s \models \ll \gg}{s[X_5] \models \ll \text{tei} \gg} \quad \dfrac{s \models \ll \gg}{s[X_6] \models \ll \text{ru} \gg}}{s[X_3] \models \ll \text{ki\_tei\_ru} \gg}}{s[X_1] \models \ll \text{zutto\_ki\_tei\_ru} \gg}$$

$$\begin{cases} X_1 = X_2 \& X_3 \\ X_2 = \sigma \prec u \\ X_3 = X_4 \& X_5 \& X_6 \\ X_4 = [\,] \\ X_5 = \sigma \prec s \text{ or } s \subset \sigma \\ X_6 = u \subset s \end{cases}$$

Figure 7: the inference tree

The *ip* and *res* correspond to in-progress state, target state, and resultant state, respectively.

Among the situations of several kinds, discourse situations are represented by object terms of the following form:

$$dsit[fov = Fov, pov = Pov, src = U]$$

where U is the object term representing the utterance situation.

Thus the propositional content of the sentence "John is running" is represented by

$$\begin{aligned} &dsit[fov = ip, pov = pres, src = u_1] : \\ &\quad inf[v\_rel = [rel = run, cls = act_2, \\ &\qquad pers = [fov = ip, pov = pres]], \\ &\qquad args = [agt = john]]. \end{aligned}$$

For simplicity, the object term $[rel = run, pers = [fov = ip, pov = pres]]$ is written as $is\_running$.

Next, the lexical entries of Japanese are defined. For example, the Japanese expression "*ki-tei-ru*" consist of three words. The lexical entries are as follows.

$$\begin{aligned} &dict :: v[cls = act_1, rel = put\_on, form = ki];; \\ &dict :: v[cls = act_2, rel = run, form = hashi];; \\ &dict :: auxv[asp = state, form = tei];; \\ &dict :: affix[pov = pres, form = ru];; \end{aligned}$$

The ambiguity is processed by the mapping from a pair of the class of a verb and the aspect of an auxilially verb:

$$\begin{aligned} &(act_1, state) \rightarrow \{ip, tar, res\}, \\ &(act_2, state) \rightarrow \{ip, tar\}, \\ &(act_3, state) \rightarrow \{tar, res\}. \end{aligned}$$

For example, the expression "*ki-tei-*" has three interpretations, while "*hashi-tei-*" has two interpretations. The target and resultant are the states after an event's having culminated. Thus, it is possible to disambiguate the interpretations if some evidence that the event has culminated or not. for example, the successive utterance of "*mi ni tuke-tei-nai*" (does not wear) makes it clear that the first sentence "*ki-tei-ru*" should be interpreted as having *ip* as its field of view.

The definition of a tiny interpreter is given in the appendix 5.

A toplevel query corresponds to the definition of the meaning of a sentence in Situation Semantics, and is of the following form:

```
?- mi[u=u1,exp=Exp,e=E,infon=Infon] ||
   {E=dsit[fov=Fov,pov=Pov,src=u1]}.
```

This query says that the meaning of the expression "**Exp**" in an utterance situation "**u1**" is represented by the described (temporal) situation "**E**" and the infon "**Infon**" where the variable "**Fov**" and "**Pov**" represent the temporal perspective.

For example, the following result is given by this interpreter:

```
?- mi[u=u1,exp=[ki,tei,ru],
      e=dsit[fov=Fov,pov=Pov,src=u1],
      infon=Infon].
```

```
Answer:
  Fov = {ip,tar,res}
  Pov = pres
  Infon =
    inf[v_rel=[rel=put_on,cls=act1,pers=P],
        args=_]
  P = [fov=Fov,pov=Pov]
```

This means that the expression "*ki-tei-ru*" has three interpretations depending on which **fov** is applied, because the verb "*ki-*" introduces all the three states[8].

On the contrary, expressions like "*hashi-tei-ru*" and "*waka-tei-ru*" has two interpretations, because those verbs can not introduce all the three states.

```
?- mi[u=u1,exp=[hashi,tei,ru],
      e=dsit[fov=Fov,pov=Pov,src=u1],
      infon=Infon].

Answer:
  Fov = {ip,res}
  Pov = pres
  Infon =
    inf[v_rel=[rel=run,cls=act1,pers=P],
        args=_]
  P = [fov=Fov,pov=Pov]

?- mi[u=u1,exp=[waka,tei,ru],
      e=dsit[fov=Fov,pov=Pov,src=u1],
      infon=Infon].

Answer:
  Fov = {tar,res}
  Pov = pres
  Infon =
    inf[v_rel=[rel=understand,cls=act1,pers=P],
        args=_]
  P = [fov=Fov,pov=Pov]
```

5 Conclusion

We have introduced the idea of temporal perspectives for situations, to explain the variety of language expressions for information in real situations. As a perspective, we assumed a topological relation between the three parameters of the standpoint of view ($\|u\|_t$), the field of view ($\|s\|_t$), and the duration of information ($\|\sigma\|_t$), each of which is the mentally recognized location of the utterance situation, the described situation, and the infon, respectively in terms of the relation theory of meaning.

We have defined tense and several important aspectual distinctions such as perfective /imperfective and durative /non-durative, with the perspective, that should be used as the partial information for the situated inference system. In addition, we tried to define other temporal features of verbs such as telic /atelic and temporal well-/ill-foundedness, to see the validity of our formalization.

Our framework for the situated inference of temporal information is to infer the whole temporal features of phrases or sentences, collecting the partial information that is carried by each lexical item, and to solve the ambiguity partial phrases they may have. We are required to have mechanisms for that system, both Prolog-like backward chaining and maintenance of consistency in modules (situations, in our case), so that we can utilize the knowledge representation language $\mathcal{QUIXOTE}$ in ICOT. We have implemented an inference system to solve the ambiguity of Japanese '*-teiru*', the $\|\sigma\|_t$ of which may refer to different parts of a deed. In that experiment, the problem is solved together with another lexical item which offers the information 'which $\|\sigma\|_t$ coincides with $\|s\|_t$'.

Our inference system is still small, and needs to be developed to cover many other kinds of lexicon and temporal ambiguity. According to this future work, we might be required to reconsider the structure of perspectives. We are still trying to determine other temporal features of verbs, and, as a task in the near future, we are going to try tto define the temporal perspectives of sentence adverbs.

Acknowledgment

Thanks to the members of STS-WG (Working Group for Situation Theory and Semantics) of ICOT for their stimulating discussions and suggestions. Special thanks to Kuniaki Mukai who gave the authors many important suggestions about the treatment of perspectives in Situation Theory, to Kazumasa Yokota who gave authors

[8] In $\mathcal{QUIXOTE}$, the fact $o[l = \{a, b\}]$ is interpreted as the two facts, $o[l = a]$ and $o[l = b]$.

many important comments, and to Koiti Hasida who explained the JPSG theory for verbs to the authors. Special thanks also to Mitsuo Ikeda of ICOT, who has studied the analysis of Japanese *-teiru*.

References

[Barwise89] J. Barwise. *The Situation in Logic.* CSLI Lecture Notes 17, 1989.

[Barwise83] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, 1983.

[Comrie76] B. Comrie. *Aspect.* Cambridge University Press, 1976.

[Cooper85] R. Cooper. Aspectual classes in situation semantics. Technical Report CSLI-84-14C, Center for the Study of Language and Information, 1985.

[Dowty79] D. Dowty. *Word Meaning and Montague Grammar.* D.Reidel, 1979.

[JPSG91] JPSG-WG. The minutes of Japanese phrase structure grammar. 1987-91.

[Allen84] J.F.Allen. Towards a general theory of action and time. *Artificial Intelligence*, 1984.

[Kamp79] H. Kamp. *Events, Instants, and Temporal References*, pages 376–417. Springer Verlag, 1979. in Semantics from Different Points of View.

[Cooper86] R.Cooper. Tense and discourse location in situation semantics. *Linguistics and Philosophy*, 9(1):17–36, February 1986.

[Tojo90] S. Tojo. A temporal representaion by a topology between situations. In *Proc. of SICONLP '90.* University of Seoul, 1990.

[Parsons90] T.Parsons. Events in the Semantics of English. MIT press, 1990.

[Yasukawa90] H. Yasukawa and K. Yokota, "Labeled Graphs as Semantics of Objects", In *Proc. SIGDBS and SIGAI of IPSJ*, Oct., 1990.

[Yasukawa92] H. Yasukawa, K. Yokota, H. Tsuda, Objects, Properties, and Modules in *Quixote*, In *Proc. FGCS'92*, Tokyo, June, 1992.

Appendix - The interpreter

```
%% Subsumption Definition
act1 >=  act2;; act1 >= act3;;

%% Lexical Entry
dict :: v[cls=act_1,rel=put_on,form=ki];;
dict :: v[cls=act_2,rel=run,form=hashi];;
dict :: v[cls=act_3,rel=understand,form=waka];;
dict :: auxv[asp=state,form=tei];;
dict :: affix[pov=pres,form=ru];;
dict :: affix[pov=past,form=ru];;

%% Top level
mi[u=U,exp=[],e=D,infon=Infon];;
mi[u=U,exp=[Exp|R],e=D,infon=Infon] <=
    d_cont[exp=Exp,e=D,infon=Infon],
    mi[u=U,exp=R,e=D,infon=Infon];;

%%%% Interpretation Rules
d_cont[exp=Exp,e=dsit[fov=Fov,pov=Pov,src=U],
       infon=inf[v_rel=V_rel,args=Args]]
  <= dict : v[cls=CLS,rel=Rel,form=Exp] ||
     {V_rel=[rel=Rel,cls=CLS,pers=P]};;
d_cont[exp=Exp,e=dsit[fov=Fov,pov=Pov,src=U],
       infon=inf[v_rel=V_rel,args=Args]]
  <= dict : auxv[asp=ASP,form=Exp],
     map[cls=CLS,asp=ASP,fov=Fov] ||
     {V_rel=[rel=_,cls=CLS,pers=P],
      P = [fov=Fov,pov=_]} ;;
d_cont[exp=Exp,e=dsit[fov=Fov,pov=Pov,src=U],
       infon=inf[v_rel=V_rel,args=Args]]
  <= dict : affix[pov=Pov,form=ru] ||
     {V_rel=[rel=_,cls=_,pers=P],
      P = [fov=_,pov=Pov]} ;;

%% Field of view Mapping
map[cls=act1,asp=state,fov={ip,tar,res}] ;;
map[cls=act2,asp=state,fov={ip,res}] ;;
map[cls=act3,asp=state,fov={tar,res}] ;;
```

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Parallel Cooperation Model for Natural Language Processing

Shigeichiro Yamasaki, Michiko Turuta, Ikuko Nagasawa, Kenji Sugiyama

Fujitsu LTD.
1015, Kamikodanaka Nakahara-Ku, Kawasaki 211, Japan

Abstract

This paper describes the result of a study of a natural language processing tool called "Laputa", which is based on parallel processing. This study was done as a part of the 5th generation computer project. The purpose of this study is to develop a software technology which integrates every part of natural language processing: morphological analysis, syntactic analysis, semantic analysis and so on, to make the best use of the special features of the Parallel Inference Machine.

To accomplish this purpose, we propose a parallel cooperation model for natural language processing that is constructed from a common processor which performs every sub-process of natural language processing in the same way. As a framework for such a common processor, we adopt a type inference system of record-like type structures similar to Hassan Ait-Kaci's psi-term [Aït-Kaci 86], Gert Smolka's sorted feature logic [Smolka 88] or Yasukawa and Yokota's object-tem [Yasukawa 90].

We found that we can utilize parallel parsing algorithms and their speed-up technology to construct our type inference system, and we then built a type inference system using an algorithm similar to a context-free chart parser. As a result of experimentation to evaluate the performance of our system on Multi-PSI, the simulator of the Parallel Inference Machine, we have been able to achieve a speed-up of a factor 13 when utilizing 32 processors of Multi-PSI.

1 Introduction

With the advance of semiconductor technologies, computers can be made smaller and cheaper, so that we can increase the value of a computer by giving it multiprocessor abilities. However, the software application technology for a parallel machine is still at an unsatisfactory level except for some special cases.

The Parallel Inference Machine which is being developed in the 5th generation computer project has some special features such as an automatic synchronization mechanism and a logic programming language allowing declarative interpretations.

Such features make complicated parallel processing tasks, that used to be practically impossible, possible to realize. Knowledge Information Processing is one such application which needs lots of computational power and consists of very complicated problems, but we expect that the Parallel Inference Machine will make these problems amenable to parallel processing. The purpose of this study is to propose a parallel cooperation model which makes natural language processing more natural by making use of the parallel inference machine features.

In this paper, we will discuss the schema of the parallel cooperation model, as well as its realization and show the experiment results of the model capacity evaluations on the Multi-PSI, the simulator of the parallel inference machine.

2 Parallel cooperation model

The advantages of using the Parallel Inference Machine lie not only in the processing speed, but also in the problem solving techniques. We were able to find more natural ways of solving a problem by looking at it from the parallel processing point of view.

In recent years, system integration has often been suggested in the field of natural language processing. This involves the integration of morphological analysis, syntactic analysis, semantic analysis and speech recognition, and the integration of analysis and generation. The implication is that the various natural processing mechanisms at every stage must be linked to each other in order to understand natural language entirely. [Hishida 91]

As the basis of this way of thinking, it is emphasized that our information processing has been carried out under "partialness of information", in other words, incompleteness of information. From the above, we can derive that a system which aims at integrating natural language processing could adopt parallel processing because it disregards the processing sequence.

We adopted a mechanism which integrates all natural language analysis processing stages and makes them cooperate in parallel as the fundamental processing model.

Also we have added a priority process as an extension, in order to improve the processing efficiency. This priority process is the combination of both load balance and the parallel priority control. We call this process 'competition' and we call the extended parallel cooperation model the "model of cooperation and competition". However, we shall not discuss 'competition' in this paper.

3 Realization of automatic parallel cooperation

It is well known that the integration of natural language processing and parallel cooperation is a natural model. However, very few systems based on this model are reported to have been actually built. One of the main problems has been modularity.

Various research projects in natural language processing have been achieving good results in the fields of morphological analysis and syntactic analysis. However, these systems were designed as independent modules and very often their interfaces are very restricted and internal information is normally invisible from the outside. To carry out efficient parallel cooperation, all processes must be able to exchange all of their information with each other. Therefore construction of methods of information exchange between the various modules and the control of these exchanges will be serious and complicated problems.

One way to solve this problem is to make an abstraction of the processing framework, so that analysis phases such as morphological analysis, syntactic analysis etc. are carried out by one single processing mechanism. One such approach is Hashida's Constraint Transformation [Hasida 90]. We have adopted an approach similar to that of among others Hashida, in the sense that all levels of processing are carried out by one and the same processing mechanism. Our processing framework, however, does not utilize Constraint Transformation, but rather Type Inferencing with respect to record-like type structures, which is comparable to Hassan Ait-Kaci's LOGIN , Gert Smolka's Feature Logic, or the Object Terms in Yasukawa and Yokota's QUIXOTE.

In our system, the usage of type inferencing can be seen to have two aspects: it works as a framework for analysis processing as well as for cooperation. Analysis processing employs a vertical kind of type judgment, as exemplified by the cooperation between morphological analysis and syntactic analysis. In morphological analysis characters are considered to be objects, and morphemes are to be taken as types; but when we perform syntactic analysis, morphemes are considered to be objects.

The usage of type inferencing as a framework for cooperation, the second aspect of this usage mentioned above, is as a means for exchanging information between objects and types and for structuring the contents of this information. Here both objects and types are represented as typed record structures containing shared or common variables, and information exchange is implemented through the unification of shared variables in two typed record structures representing an object and a type.

This unification mechanism of typed record structures has a mechanism to judge the types of objects that are instantiated to field elements through communication, and this is what was called the vertical type judgment mechanism.

Parallel cooperation between syntactic and semantic processing is expressed through the unification mechanism of typed record structures.

Even if we treat all phases of natural language processing as similar in kind, it is still natural, for the sake of ease of grammar development and debugging, to do the development of the distinct processing phases separately. For this reason, we have structured our system so that concept organization rules for morphological, syntactic and semantic processing can be developed separately. Parallel cooperation is then realized automatically by merging these diverse rules and definitions.

4 The realization of parallel analysis processing

4.1 Type judgment mechanism

Efficient algorithms exist for morphological and syntactic processing, and we cannot afford to ignore such knowledge in developing a practical system, even in the case of an integrated natural language processing system. Luckily we have found that there is a strict correspondence between our vertical type judgment and known syntactic analysis methods. Matsumoto's parallel syntactic analysis system PAX [Matsumoto 86] performs syntactic processing in parallel through a method called the "layered stream method", which is an efficient processing mechanism for search problems involving parallel logic programming languages.

PAX employs what is basically a chart parsing algorithm. Our vertical type judgment processing formalism involves a reversal of the relationship between process and communication data in PAX. A syntactic analysis system using a similar processing method to ours is being considered by Icot's Taki [Sato 90]. Whereas PAX is strongly concerned with the clause indexing mechanism of logic programming languages, our method concentrates on increasing OR-parallellism and reducing the amount of data communication in parallel execution.

How we interpret phrase structure rules, using the type ordering relation "<" and type variables, is shown below.

```
s <- np,vp
```

This is rewritten based on the rightmost element as follows.

```
vp < (np -> s)
```

Here the ordering relation "<" expresses a superordinate-subordinate relationship between types. Intuitively this means that the object that is judged to be a subordinate type can also be judged to be a superordinate type. It follows that the meaning of this rule is that the object that can be judged to be the vp, can also be judged to be a function of type np to s.

```
s <- advp,np,vp
```

In a case like this one, we embed functions to produce the following.

```
vp < (np -> (advp -> s))
```

When there are several possibilities, this is expressed in a direct sum format as follows.

```
vp < (np -> ((advp -> s) + s))
```

The dictionary is a collection of type declarations as follows.

```
(in,the,end):advp
love:np
wins:vp
```

Analysis is executed as a process of type judgment of a word string. In other words, analysis is the execution of the judgment of a type assignment such as the one below.

```
(in,the,end,love,wins):s
```

The execution is bottom-up. First the type of every word is looked up among the type declarations. The words then send these type judgments to their right adjacent element. If these types again have superordinate types, then they are treated as follows.

If the superordinate type is a function, then a process is generated which checks the possibility that the typed object received from the left is appropriate. If it is not a function (in which case it is atomic), then this type judgment formula is sent to the right adjacent element, and also it is checked whether it has a superordinate type. When the result of a superordinate type or function appropriateness is a direct sum, then this result is handled in OR-parallel form. Repeating this kind of processing over and over, we get as answers all the combinations of elements from the leftmost to the rightmost that satisfy "s".

One of the special features of this processing formalism is that, when sending an object of atomic type, the pointer to the position of the leftmost of the elements that make up this object is sent along as the "exit of communication path". Hereby the partial tree that is constructed upon reception of this object in fact is capable of including all atomic-type objects that are structured to the left of the received object. If we translate this to structure sharing in sequential computation, we see that we can avoid unnecessarily repeating the same computation while retaining the computational efficiency of a chart parsing algorithm for context-free grammars.

Below we give the KL1 program for the fundamental part of vertical type judgment. Note however that the notation we have used above is transformed to KL1 notation, in the manner explained directly below.

```
direct sum
     type+,...,+type ==> [type,...,type]
type declaration
     object:type  ==> type(object,T) :-
                                 true| T=type.
type ordering
    type < type   ==> upper(type,T) :-
                                 true| T=type.
input format
  (in,the,end,love,wins):s
   ==>judgment([in,the,end,love,wins],s,R).
```

Note: R will contain the result of computation

Also we use "*" for the operator that constructs the pair of the sending atomic type and the stream, and the atom "Leftmost" as an identifier for the leftmost position of the input.

```
judgment(Objects,Type,Result) :- true|
        objects(Objects,'Leftmost',R),
        judged_as(R,Type,Result).

objects([],L,R) :- true|L=R.
objects([Word|Z],L,R) :- true|
        type(Word,Type),
        sum_type(Type,L,R1),
        objects(Z,[Word|R1],R).

sum_type([],L,R) :- true|L=R.
sum_type([Type -> Type2|Z],L,R) :- true|
        function_type(Type ->Type2,L,R1),
        sum_type(Z,L,R2),
        merge({R1,R2},R).
otherwise.
sum_type([Type|Z],L,R) :- true|
        atomic_type(Type,L,R1),
        sum_type(Z,L,R2),
        merge({R1,R2},R).

function_type(Type -> Type2,[],R) :-
        true|R=[].
function_type(Type -> Type2,'Leftmost',R) :-
        true|R=[].
function_type(Type-> Type2,[Type *L1|L],R) :-
```

```
        true|
        sum_type(Type2,L1,R1),
        function_type(Type -> Type2,L,R2),
        merge({R1,R2},R).
otherwise.
function_type(Type -> Type2,[_|L],R) :-
        true|
        function_type(Type -> Type2,L,R).

atomic_type(Type,L,R) :-
        true|
        upper(Type,Upper_Type),
        R=[Type*L|R1],
        sum_type(Upper_Type,L,R1).

judged_as([],Type,Result) :-
        true|Result=[].
judged_as([Type*'Leftmost'|L],Type,Result) :-
        true|
        Result=[Type|R],
        judged_as(L,Type,R).
otherwise.
judged_as([_|L],Type,Result) :- true|
        judged_as(L,Type,Result).

% Example of dictionary:

type(love,Type) :- true|
        Type=[np].
type(wins,Type) :- true|
        Type=[vp].
type(end,Type) :- true|
        Type=[the -> [in -> [advp]]].

% Example of grammar:

upper(vp,Upper_Type) :- true|
       Upper_Type=[np -> [advp -> [s], s]].
```

4.2 Unification mechanism of the record-like type structure

A record-like type structure is a pair of a sort symbol and a description. A sort symbol denotes the sort to which the type belongs. A description is a so-called record structure formed by pairs of feature names and their values. The feature value is also a description or an object. However a description is unlike an ordinary record structure in that its feature and value pairs are not always apparent. Indeed, the purpose of this structure is to obtain incremental precision from partial information, just like the feature structures used in unification grammar formalisms such as LFG.

In systems like Ait-Kaci's psi-term, Smolka's sorted feature structure or Yasukawa & Yokota's object term, the value of a feature is also a record-like type structure. However in our system, a value of a feature is not a record-like type structure but a description and only the terminal nodes of a feature structure tree are typed objects. This is to improve the efficiency of calculation.

In our system an object is represented as a pair of a record-like type structure and an identifier of the object. The value of a feature can be a variable. However the unification of descriptions involves merging feature structures rather than instantiating variables.

Variables of feature value play the role of a tag for the merging point in feature unification. In our system such variables also play the role of communication pass to exchange information for our parallel cooperation. Sometimes a variable can be assigned a type. When a variable is assigned a type such a typed variable must be instantiated by an object.

Below we give an example of a record-like type structure.

```
{human,[parents=[father={human,211,[name=taro]},
                 mother=X:{human,[]}]]}
```

This example shows a type which is sorted "human" and satisfies some constraints as a description. The description has a feature "parents" and the value of the feature is also a description that contains the feature of "father" and feature of "mother". The value of the feature "father" is an object that is of sort "human" and named "taro" and its object identifier is "211". The value of feature "mother" is a typed variable. The type of the variable is sorted "human" and its description has no information.

The unification mechanism for the record-like type structure is realized as the addition of information to the table of the pairs of the tag and the structure to which the tag referred. The unification process is the merging process to construct the details of the record-like type structure. When the typed variable is instantiated by an object, the type judgment process is invoked.

This is in concreto how our parallel cooperation mechanism for semantic analysis and syntactic analysis works.

4.2.1 Parallel cooperation and record-like type structure

A type can be seen as a program which can process an object. This implies that there is a close relation between merging of information using record-like type structures on the one hand, and the "living link" between objects or programs on the other. As an example, imagine that a graph object which was created by a spread sheet program is passed on to an object which is a word processor document. If we want such a graph object to be a "living" object, re-computable by the creator program, then it must be annotated by its creator as a data type in the record structure of the word processor document. Now

when the data is re-computed, the system will invoke its creator application program automatically. The type theory of record-like type structures can be viewed as a framework for this kind of cooperation of different application programs.

Laputa's principle of automatic parallel cooperation is a parallel version of this "live linking". Vertical type judgment of morphological analysis and syntactic analysis is an application program in this sense.

We can extend this live linking even further by using variables that are shared between objects and types, so that we can propagate information to objects within objects. For example, if a graph object from a spread sheet is pasted to a word processor document, and some of the data within the graph is shared with a part of the document text, then re-computation of the spread sheet program will happen when that part of the document text is modified. In our system, such re-computation is realized by communication of processes.

5 The grammar and lexicon for parallel cooperation

In this section we explain the syntax and description method for the grammar and lexicon for our system. Grammar rules and lexical items are described as a type definition or an ordinal relation of types. Our parallel processing mechanism treats morphological data and syntactic data in a uniform way. However it is not efficient to use exactly the same algorithm for morphological processing as for syntactic processing, because morphological processing examines only immediately adjacent items and therefore does not need context-free grammar. Our processing mechanism treats characters and morphemes in a slightly different way. For this reason characters and morphemes are distinguished as data types. Another, more essential reason for this is the problem posed by morphemes that consist of only one character. If there is no difference between a character and a morpheme, then our type judgement process will never be able to stop.

5.1 Some examples of grammatical and lexical description

5.1.1 Dynamic determination of semantic relation of subject and object

The semantic categories of subject and object are not determined only by the verb with which they belong. In many, if not most cases, the adequacy of the semantic category of the object changes according to the subject. Because of this, the required semantic categories of subject and object should not be fixed in the lexical description of the verb.

The example grammar rules below show how the adequacy of the semantic category of the grammatical object can vary dynamically depending on the subject.

```
{np,[sem=Ob]} < ({vt,VT} -> {vp,VT=[obj=Ob]})
{vp,VP} < ({np,[sem=Ag]} -> {s,VP=[agent=Ag]})
```

In this example the first grammar rule shows that the superordinate type of the type "np" is a function of type "$vt- > vp$". This rule means that an object which is judged as the type "np" is also a function which, if applied to an object of type "vt", results in an object of type "vp". In this rule every description of "vt" is merged to "vp" and the value of the feature "sem" of "np" is unified with the value of feature "obj" of the type "vp".

The next grammar rule means that an object of type "vp" is also a function of type "$np- > s$". This rule means that the value of the feature "sem" of subject "np" is unified with the value of the feature "agent" of "vp".

Now we also show some lexicon entries to go with these rules.

```
eats:{vt,[agent=Ag:{animal,[eat_obj=Ob]},obj=Ob]}
john:{np,[sem={human,Id,[name='John']}]}
the_tiger :{np,[sem={tiger,Id,[]}]}
```

In the lexicon the object "eats" has type "vt" and complex description. In the description the value of the feature "agent" is a typed variable and the type of the variable is sorted "animal" and the value of the feature of "eat-obj" is unified with the value of the feature of "obj" of type "vp".

The rules specifying semantic categories look as follows.

```
{tiger,[]} < {animal,[eat_obj=E:{animal,[]}]}
{human,[]} < {animal,[eat_obj=E:{food,[]}]}
```

These rules mean that a tiger is an animal which eats animals and a human is an animal which eats food, respectively.

Although under these rules of grammar, lexicon and semantic categories 'john' and 'the_tiger' are both animals, the judgment (the_tiger,eats,john):s succeeds but (john,eats,the_tiger):s fails, because John is a human and a human is an animal which eats food but a tiger cannot be judged as food from the rules governing semantic categories.

5.1.2 Subcategorization

The next example is the lexical entry for the Japanese verb "hanasu"(to speak). This verb is subcategorized by 3 "np"s which are marked for case by the particles "ga", "wo" and "ni".

```
hanasu:{vp,[subcat=Case:{ga,wo,ni},
            predicate=[ga=[gram_rel=subj,
                           sem=G],
                       wo=[gram_rel=comp,
                           sem=W],
                       ni=[gram_rel=obj,
                           sem=N],
            sem=[rel=speak,
                 agent=G:{human,[]},
                 object=N:{human,[]},
                 topic=W:{event,[]}]]]}.
```

In addition to the above, suppose that we also have the following lexical entries and grammar rules.

```
ga:{noun,N} -> {np,N=[case_marker=ga]}
ni:{noun,N} -> {np,N=[case_marker=no]}
wo:{noun,N} -> {np,N=[case_marker=wo]}

{vp,VP=[subcat=Case:SUB]} <
       {np,[case_marker=Case]} ->
             {vp,VP=[subcat=New:SUB-{Case}]}
```

In that case the type judgments for the sentences below will be successful.

```
(john,ga,mary,ni,anokoto,wo,hanasu):{vp,[]}
(mary,ni,anokoto,wo,john,ga,hanasu):{vp,[]}
```

5.1.3 Example of the conceptual system rules

The conceptual system rules are sets of rules which determine superordinate and subordinate relations of concepts. The semantic analysis of Laputa uses these conceptual rules when it performs semantic judgemant.

```
{object,[]} < {'Top',[]}
{event,[]} < {'Top',[]}
{concrete-object,[]} < {object,[]}
{creature,[]} < {concrete-object,[]}
{human,[]} < {creature,[]}
{student,[]} < {human,[]}
```

6 An experiment using Laputa

6.1 Conditions of the experiment

computer Multi-PSI 32PE construction

OS PIMOS 3.0.1

The size of the grammar and dictionary

grammar rules	651
words	14,613
morphemes	8,268
concepts	770

We used the syntactic grammar and morphological grammar which were developed by Sano of ICOT's 6th Laboratory [Sano 91]. We made the conceptual system rules in accordance with the conceptual system of the Japan Electronic Dictionary Research Institute EDR.

The experiment We used 22 test sentences and examined 3 types of cooperation pattern: (1) syntactic analysis only, (2) cooperation of morphological analysis and syntactic analysis, and (3) cooperation of morphological analysis, syntactic analysis and semantic analysis.

We checked the relation between the number of processor elements utilized and the number of reductions and processing time for each of these 3 cases.

All the tests have been performed three times, and the measurements given here are the averages computed from these three processing runs.

Example of analysis result To indicate the level of processing of this experiment, I will show the result of analysis of a example sentence.

Example sentence

「彼が父の事業を継いだ」

(He inherited his father's business.)

Analysis result

```
vp(1,[subcat=SUB:[],
      infl=u_ga,
      predicate=[
           lex=継,
           soa=[ga=[sem={man,1,[]},
                    gram_rel=subj],
                wo=[sem={job,6,
                         [of_type={man,2,[]}]},
                    gram_rel=comp],
                sem={tugu,8,
                     [agent={man,1,[]},
                      object={job,6,
                              [of_type=
                                   {man,2,[]}]}]]
                tenseless=action],
           polarity_of_soa=true,
           judgment=affirmation,
           aspect=not_continuous],
      mood=finished,
      recognition=[modality=descriptive,
                   acceptance=affirmative]])
```

6.2 Outcome of the experiment

The the following graph shows, for the analysis of example sentence 12, how the speed-up ratio changes as the number of processors is increased from 1 to 32.

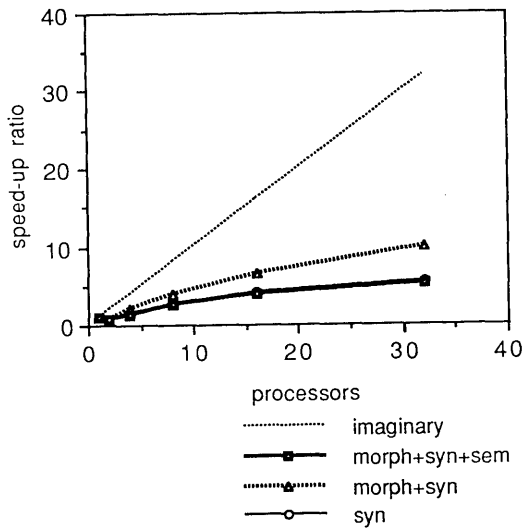


Figure 1: Example 12 processors and speed up ratio

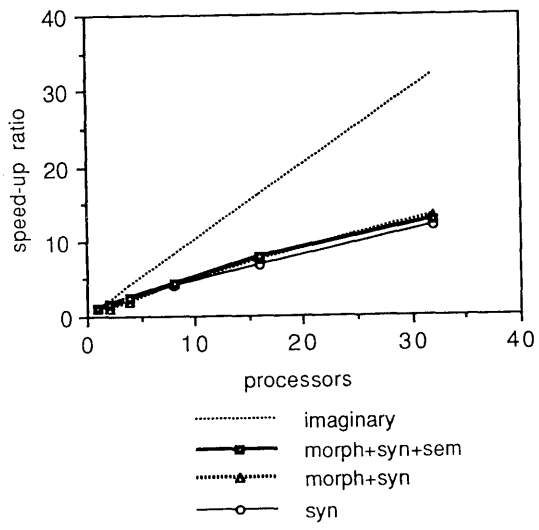


Figure 2: Example 14 processors and speed up ratio

The behavior of the cooperative process of morphological, syntactic and semantic analysis is almost identical to that of syntactic analysis alone, while the cooperation of just morphological and syntactic analysis shows a much better speed up ratio relatively. This might lead one to think that cooperation of just morphological and syntactic analysis makes for a better speed-up ratio. However examples involving a greater amount of calculation do not show this difference. Figure 2 is the result of the analysis experiment on example sentence 14.

This graph show that all three types of cooperation have the same speed-up ratio, which is a different result than that we deduced from example sentence 12.

We can interpret this difference as resulting from a difference in the amount of calculation. Both the syntax-only calculation and the cooperative syntactic and morphological analysis process for example sentence 12 simply do not involve enough computation to fully show the potential speed-up ratio. Sentence 14, on the other hand, requires enough computation for any of the three types of cooperation, so that we can more clearly see the speed-up ratio.

To verify this assumption, we plot the speed-up ratio against the amount of calculation for the three types of cooperation. As the graph shows, all three cooperation types show similar behavior for this relation. We can understand why this is so if we recall that in our system, all modes of processing employ the same basic processing mechanism.

In the graph, we can see that the speed-up ratio rises steeply while the number of reductions remains small, but gradually becomes saturated as the number of reductions grows.

In the bar graph of Figure 4, we can see that the number of reductions for sentence 12 in the case of cooperative morphological and syntactic analysis is about 1,200,000, while in the other two cases it is about half as much (approximately 600,000). Because the number of reductions as a whole is small, this difference is important. Example 14 on the other hand involves enough computation so that the effect is minimized and the similarities between the processing modalities are allowed to come out.

7 Conclusion

In this paper, we proposed a model for integrated natural language processing on a parallel inference machine. This model is realized by choosing similar processing schemes for morphological, syntactic and semantic analysis and having these cooperate in parallel.

Also, we have carried out an experiment to evaluate the practicality of our processing model.

As the result of our experiment we have been able to realize speed-up to a factor of about 13 when utilizing 32 processor elements.

The results also showed that the speed-up ratio is determined only by the amount of computation, and is not influenced by the configuration of cooperating analysis processes.

If our processing model is to be practical as a method for a real parallel inference machine, the object of analysis should require a great amount of calculation because when the amount of calculation is low we can not expect a satisfactory speed-up ratio.

We think that our processing model has the potential

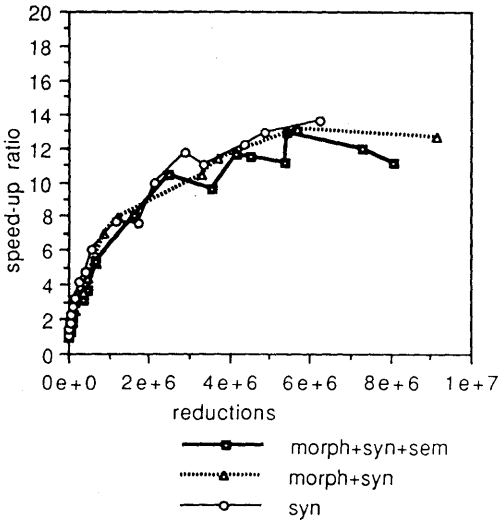


Figure 3: reductions and speed-up ratio

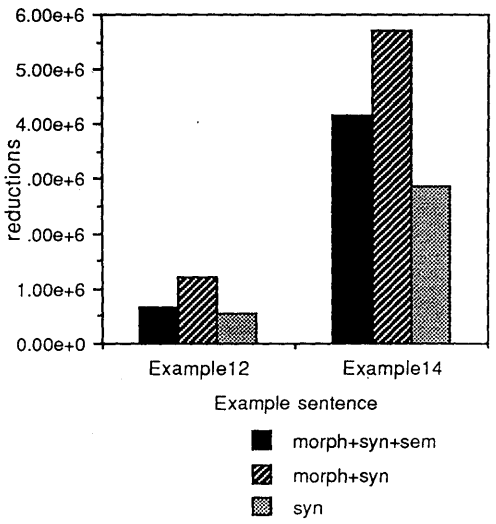


Figure 4: cooperation case and reductions

to be a practical technology for natural language processing, and that it can help increase the amount of cooperation with fields like pragmatics, speech recognition and the utilization of world knowledge.

Acknowledgments

We would like to thank Yuuichi Tanaka, Hiroshi Sano and other members of 6th laboratory of ICOT.

We were supported by Hirosi Onodera, Naoto Hirota, Yoshiko Kamimura and other members of Fujitsu-FIP Ltd in our experiments.

We are also grateful to Eric Visser for his support in finalizing this paper.

References

[Martin-Löf 84] Per Martin-Löf : Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes, 1984.

[Aït-Kaci 86] Hassan Aït-Kaci and Roger Nasr, LOGIN: A Logic Programming Language with Built-in Inheritance, The Journal of LOGIC PROGRAMMING, Vol. 3, No. 3, Oct. 1986.

[Schmidt-Schauss 89] M. Schmidt-Schauß, Computational Aspects of an Order-Sorted Logic with Term Declarations, Lecture Notes in Artificial Intelligence, Springer-Verlag, 1989.

[Smolka 88] Gert Smolka, A Feature Logic with Subsorts, IBM Deutschland, Stuttgart, West Germany, LILOG Report 33, May 1988.

[Yasukawa 90] Hideki Yasukawa, Kazumasa Yokota, The Overview of a Knowledge Representation Language Quixote, ICOT (draft), Oct. 21, 1990.

[Sano 91] Sano Hiroshi, User's Guide to SFTB, ICOT (draft), Sep. 1991.

[Hasida 91] Koiti Hasida, Aspects of Integration in Natural Language Processing (In Japanese), Japan Society for Software Science and Technology, COMPUTER SOFTWARE, Vol. 8, No. 6, Nov. 1991.

[Hasida 90] Koiti Hasida, Sentence Processing as Constraint Transformation, Proceedings of 9th European Conference on Artificial Intelligence, 1990

[Matsumoto 86] Yuuji Matsumoto, A Parallel Parsing System for Natural Language Analysis, Proc. of 3rd Internatioanl Conference on Logic Programming, London, 1986.

[Sato 90] Hiroyuki Sato, An improvement of a parallel natural language analyzing system PAX (In Japanese)), Proc. of KL1 Programming Workshop '90, ICOT, Tokyo, 1990.

Appendix

Example sentences

Example1 太ってやる
(I will get fat.)

Example2 木に竹を継ぐな
(Do not connect bamboo to wood.)

Example3 彼は父の事業を継いだ
(He inherited his father's business.)

Example4 話の途中で電話を切るな
(Don't hang up the telephone in the middle of a conversation.)

Example5 今日の奈々子はこの前より太っていた
(Today's Nanako is fatter than before.)

Example6 部屋から出たら電気のスイッチを切りなさい
(Turn off the light when you leave the room.)

Example7 彼を呼んで小言を言ったから順調に進んでいる
(Because I called and scolded him, it is progressing well.)

Example8 そういう理由で髪を切ったことを彼女は話しはじめた
(She began to tell me that that was the reason why she cut her hair.)

Example9 東京に行く彼女が乗った汽車が進みはじめたら雪が降ってきた
(When the train that she, who was going to Tokyo, was on started moving, snow began to fall.)

Example10 私は彼女を信じていたのにそういう奈々子は夜までわざと帰ってこなかった
(Though I believed her, Nanako didn't return until the night on purpose.)

Example11 その会議で私だけが意見を言ったことで後で会社でこういう問題になった
(Because only I expressed my opinion at that meeting, later at the company we had this kind of problem.)

Example12 電車が来るのを待ちながらパーティに呼ぶ友達のことを奈々子が彼に話していた
(While waiting for the train, Nanako told him about the friends that she would invite to the party.)

Example13 私の父が問題の荷物を解いて彼に見せたら後でサウジアラビア王国の女性から電話が掛かってきた
(First my father opened the package in question and showed it to him, and later we got a telephone call from a woman from the Kingdom of Saudi Arabia.)

Example14 クリスマスの夜に学校から帰ってくる奈々子を待ちながら父と話していたら雪になった
(As I talked to my father while waiting for Nanako to come home from school on Christmas Eve, it started snowing.)

Example15 アメリカからこの前に話していた専務を呼んだからあの伯父が父の事業を継ぐだろう
(Since he called that managing director I was telling you about before from America, that uncle will probably inherit my father's business.)

Example16 私だけがその方程式を解いていなかったから解き方を知っている友達に教わってその問題を解いた
(Since only I hadn't solved that equation, my friends who knew how to solve it helped me out and I solved the problem.)

Example17 私だけがその方程式を解いていなかったから解き方を知っている友達に教わってその問題を解いて寝た
(Since only I hadn't solved that equation, my friends who knew how to solve it helped me out and I solved the problem and went to bed.)

Example18 社長がアメリカからこの前に話していた専務を呼んだからあの伯父が父の事業を継ぐだろう
(Since the president called that managing director I was telling you about before from America, that uncle will probably inherit my father's business.)

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Architecture and Implementation of PIM/p

Kouichi KUMON Akira ASATO Susumu ARAI
Tsuyoshi SHINOGI Akira HATTORI
Fujitsu Limited
1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Hiroyoshi HATAZAWA
Fujitsu Social Science Laboratory Ltd.

Kiyoshi HIRANO
Institute for New Generation Computer Technology

Abstract

In the FGCS project, we have developed a parallel inference machine, PIM/p, as a one of the final outputs of the project [Taki 1992]. PIM/p has up to 512 processing elements(PEs) using two level hardware structures. Each PE has a local memory and a cache system to reduce bus traffic. The special cache control instructions and the macro-call mechanism reduce the common bus traffic, which may become the performance bottle-neck for shared-memory multi-processor systems. Eight PEs and a main memory are connected by common bus using the parallel cache protocol, we call it a *cluster*. PIM/p system consists sixty-four clusters, those are connected by dual sixth-order hyper-cube networks.

The KL1 processing system on PIM/p has two component, the compiler and the run-time support routines. The compiler uses the templates to generate PIM/p native codes from KL1-B codes. Each KL1-B instruction has a corresponding template. The codes are optimized after the expansion from KL1-B to native codes. The run-time support routines are placed in the internal-instruction memory, in the local-memory, or in the shared memory according to their calling frequencies.

The preliminary evaluation results are presented. Corresponding to the hierarchy of PIM/p, two different configuration systems: the network connected system and the common bus connected system, are compared.

The results show that the speedup ratio compared to one PE is nearly equal to the number of PEs for both configuration systems. Hence, the bus traffic is not a performance bottle-neck in PIM/p, and the automatic load-balancing mechanism appropriately distributes loads among PEs within a cluster at the evaluation.

1 Introduction

A parallel inference machine prototype(PIM/p) is now being used. It is tailored to KL1 [Ueda and Chikayama 1990], and includes up to 512 processors. A two-level hierarchical structure is being used in the new system: a processing element and a cluster(Figure 1).

Eight processing elements form a cluster, which communicates with a shared memory through a common bus using snooping cache protocols. The clusters are connected with dual hypercube packet switching networks through network interface co-processors and packet routers. The chassis consists of four clusters. The maximum PIM/p system includes sixteen chassis. A single clock is delivered to all processing elements, maintaining the phase between different chassis.

Some of the features introduced in the PIM/p system are:

- Two level hierarchical structure to allow parallel programming with common memory and to facilitate system expansion with the hypercube network.
- The macro-call instructions which have the advantages of both hard-wired RISC computers and micro-programmable instruction set computers.
- Architectural support for incremental garbage collection *Multiple Reference Bit*(MRB), which reduces memory consumption when the executing parallel logic programming languages such as KL1.
- Each processing element has a local memory, which can reduce bus traffic if the accessed data are placed in the local memory.
- Coherent cache and dedicated cache commands for KL1 parallel execution, which can also reduce common bus traffic.
- Generating the native instruction codes from intermediate KL1-B codes by optimizing compiler with a optimizer.
- The optimizer analyses data-flow for both the tag parts and the data parts independently, which can eliminate unnecessary tag operations.

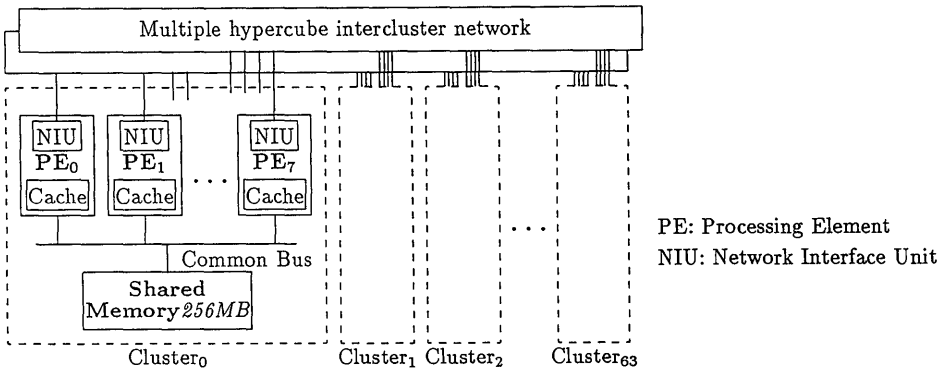


Figure 1: PIM/p system configuration

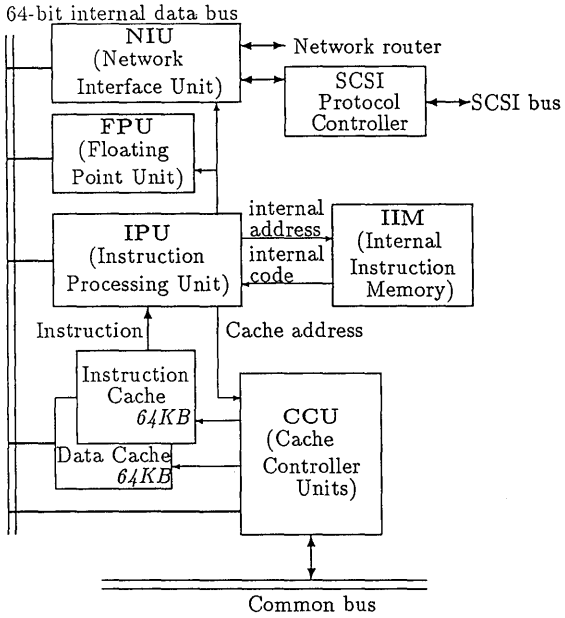


Figure 2: PIM processing element configuration

The *processing Element*(PE) consists of an *Instruction Processing Unit*(IPU), a *Cache Control Unit*(CCU) and network interface unit(NIU). Figure 2 is a schematic diagram of a PE.

In this paper, the hardware architecture and the KL1 processing system are described. In Section 2 to Section 4 we describe IPU, cache and the network system. Then, the run-time support routines for KL1, and the KL1-B compiler code generation and its optimization are described in Section5. Finally in Section 6 a preliminary performance evaluation results are presented.

2 IPU Architecture

The instruction processing unit(IPU) executes RISC-like instructions which have been tailored to KL1 execution. The instruction set has many features which facilitate efficient KL1 program execution. In this section, we describe these features.

2.1 Tagged data and type checking

To execute KL1 programs, a dynamic data type checking mechanism is needed to provide:

- Transparent pointer dereferencing.
- Polymorphic operations for data types.
- Incremental garbage collection support.

Dereference is required at the beginning of most unification operations in KL1. In dereference, a register is first tested to see whether its content is an indirect pointer or not. If it is an indirect pointer, the cell pointed to is fetched into the register and its data type is tested again.

Many operations in KL1 include run-time data type checks even after dereferencing has been completed. Unifications include polymorphic operations for data whose type is not known until run-time.

In addition, incremental garbage collection by MRB is embedded in dereferencing(See Section 2.5 for details).

Therefore, tagged architecture is indispensable for the KL1 processing. In PIM/p, data is represented as 40-bit (8-bit tag + 32-bit data), and the general-purpose register has both a data part and a tag part. The MRB is assigned in one bit of the 8 bit tag.

The tag conditions are specified as bit-wide logical operations between the tag of a register and the 8-bit immediate tag value in the instruction. An instruction can specify the logical operation as *AND*, *OR*, or *XOR* or a negations of one of these.

If an instruction specifies *XOR* as its logical operation, it checks whether the tag of the register matches the immediate value supplied in the instruction. *Xor-mask* operation does this matching under the immediate mask supplied in the instruction, which enables various groups of data types to be specified in a conditional instruction if the data types are appropriately assigned to tag bits (See Section 5.1 for details).

Various hardware flags, like the condition code of ALU operations or hardware exception flags, can be checked as the tags of dedicated registers, so these flags can be examined by a method similar to data type checking.

2.2 Instructions and pipeline execution

The processing element uses an instruction buffer and a four-stage pipeline, **D A T B**, to attempt to issue and complete an instruction. Table 1 shows the pipeline stages in ALU, memory access and branch instructions. All instructions except co-processor instructions are issued in every cycle.

Basic instructions such as ALU operations have three operands, and memory accessing instructions are limited to *load* and *store* type instructions. Pipeline execution tends to make the branch penalty large. In PIM/p, the target instruction starts four clock after the branch instruction starts. To reduce the branch penalty, *delayed branch* instructions are used. These have one delay slot after them.

The *skip* instruction is also useful. This nullifies a subsequent instruction if the skip condition is met. The skip instruction does not cause a pipeline break, so its use results in efficient instruction execution. Figure 3 shows the pipeline stages in conditional branch/delayed-branch/skip instructions.

In the PIM/p pipeline, all instructions write their results at the **B** stage and ALU or memory write instructions require source operands at the beginning of the **B** stage. The bypass from the **B** stage can eliminate interlocks. Conditional branch instructions test the condition at the **B** stage, the bypass also eliminates condition test interlocks. However, when the register is used by address calculation at the **A** stage when the value of the register has just been changed, an interlock may occur even if a bypass from **B** to **A** is prepared. Figure 4 shows this address calculation interlock. The compiler must recognize such interlock conditions and should eliminate them as far as possible.(See section 5.2.3)

2.3 Macro call and internal instructions

A RISC or RISC-like instruction set has advantages in both low hardware design cost and fast execution pipelining. However, naive expansion of KL1-B to low-level RISC instructions produces a very large compiled code.

When conditional branch is taken: condition tested at **B**

```
D A T B                 : cond. branch instruction
  D A T  canceled       : next external instruction
    D A  canceled       : 2nd external instruction
      D  canceled       : 3rd external instruction
         D A T          : branch target instruction
```

When delayed branch is used: condition tested at **B**

```
D A T B                 : cond. branch instruction
  D A T B               : next external instruction
    D A  canceled       : 2nd external instruction
      D  canceled       : 3rd external instruction
         D A T          : branch target instruction
```

When conditional skip is taken: condition tested at **B**

```
D A T B                 : cond. skip instruction
  D A T  canceled       : next external instruction
    D A T B             : 2nd external instruction
      D A T B           : 3rd external instruction
```

Figure 3: Pipeline stages of conditional branch/skip instructions

```
D A T B                 : register write instruction.
  D D D A T B           : inter-lock occurs
          D A T         : next instruction
```

Figure 4: Interlock caused by address calculation

This may cause frequent instruction cache miss-hits and may fill up the common bus band width with instruction feed, especially in tightly-coupled multiprocessors such as a PIM/p cluster. Here, reducing common bus traffic is a most important design issue as is reducing the cache miss-hit ratio. On the other hand, the static code size can be small in a high-level instruction set computer with micro-programs, such as PSI.

To meet both requirements, the processing element of PIM/p has two kinds of instruction streams, *external* and *internal*. External instructions are mostly RISC-like instructions with KL1 tag support[Shinogi et al. 1988]. Internal instructions are fed from internal instruction memory like micro-instructions.

The external instruction set includes macro-call instructions, which first test the data type of a register given as an operand, then invoke programs in the internal instruction memory(IIM) or simply execute the next external instruction, depending on the test result. Every time a macro-call instruction is executed, the corresponding macro-body instruction is fetched from IIM to reduce the calling overhead, but it is not executed unless a macro-call test condition is met (See the **S** and **C** stages of Table 1). Figure 5 shows the pipeline stages of macro-call instructions. A macro-call instruction can be regarded as a light-weight conditional subroutine call or

Table 1: Pipeline stages of ALU, memory access and branch instructions

	ALU operation	*Memory access*	*Branch*
(S)	Set IIM address, *valid only for m-call or internal instructions*		
(C)	Fetch instruction from IIM, *valid only for m-call or internal instructions*		
D	Decode	Decode / Register read for address	Decode / Register read for address
A	—	Memory address calculation	Branch address calculation
T	Register read	Cache tag access	Cache tag access
B	ALU operation / Register write	Cache data access / Register write	Cache data access / Condition test

When the condition met: condition test at **A**

```
D  A                          : macro-call instruction
   D  canceled                : next external instruction
S  C  D  A  T  B              : first internal instruction
   S  C  D  A  T  B           : 2nd internal instruction
```

When the condition is not met: condition test at **A**

```
D  A                          : macro-call instruction
   D  A  T  B                 : next external instruction
      D  A  T  B              : 2nd external instruction
```

Figure 5: Pipeline stages of macro-call instructions

as a high-level instruction with data type checking.

To reduce the overhead of passing parameters from a macro-call instruction to the macro-body, the PIM/p processing element has three *indirect registers*. The indirect registers are pseudo registers whose real register numbers are obtained from the corresponding macro-call instruction parameters.

These mechanisms may appear to be similar to those of conventional micro-programmable computers. Programs stored in IIM are written by system designers into internal instruction memory, like micro-programs. However, the internal instruction set is almost the same as the external instruction set, so a designer can use same development tools to generate both external and internal programs. Therefore, system designers can specify internal or external at the machine-language level, without writing complicated micro-instructions, as in conventional micro-programmable computers.

2.4 Dynamic test stage change

As discussed in the Section 2.3, internal instruction executions require an additional two pipeline stages, **S** and **C**, before the **D** stage, internal conditional branch causes a five clock cycle branch penalty when the branch is taken. In the case of an external branch instruction, target instruction fetch starts at **A** as an operand and the fetch finishes at the **B** stage, thus testing the condition before the **B** stage cannot reduce branch penalty.

However, internal instructions must use the **S** and **C** pipeline stages to fetch the target internal instruction. It cannot not start before the condition test. If the branch condition is determined earlier, say at stage **A**, target fetch can be started earlier. This reduces the branch penalty. However, an early condition test causes interlocking, which is common to memory address calculation, and this will occur even if the branch is not taken. Table 2 shows the advantages and disadvantages of both **B** stage and **A** stage condition tests. Some sample codings show internal conditional branches are often placed just after memory read or ALU operation instructions, and it is hard to insert non-related instructions between them. To minimize pipeline stall, an **A** stage test should be used if the previous instruction does not interlock the condition test, otherwise **B** stage test should be used.

Table 2: The advantages and disadvantages of **B** and **A** condition check

Test stage	*Advantages*	*Disadvantages*
B	No interlock	5τ branch penalty
A	1τ branch penalty	$0/1/2\tau$ interlock

1τ=1 clock cycle

Preparing two sets of branch instructions, a **B** stage test and an **A** stage test, adds instructions to the PIM/p instruction set, because the PIM/p instruction set has many conditional branch instructions for various tag checking.

Without adding instructions, the PIM/p pipeline controller decides between internal conditional branch **A** or **B**[Asato et al. 1991]. When some instructions interlock the test stage **A** of a successive internal conditional branch, the test stage is changed to **B** to avoid interlock, otherwise the test is done at **A** stage. We call this a *dynamic conditional branch test stage change*. If a compiler or a programmer can put two or more instructions between a register write instruction and a conditional branch based on the register, the test is done at the **A** stage.

2.5 MRB support

Incremental garbage collection support is one of the most important issues in parallel inference machines. The PIM/p instruction set includes several instructions for efficient execution of MRB garbage collection[Chikayama and Kimura 1987].

Using the MRB incremental garbage collection, value cells or structures are allocated from free lists, and when those allocated areas are reclaimed, the areas are linked to free lists. To support these free list operations, the *push* and *pop* instructions are used.

The MRB of each pointer and data object has to be maintained in all unification instructions. Especially in dereference, the MRB of the dereferenced result is off if and only if MRBs of both the pointer on a register and the pointed cell are MRB-off. MRB is assigned to one of the eight bit tag data. MRB-on means the bit is 1, MRB-off means 0 respectively. Therefore logical *or* of both the pointer MRB bit and the pointed data MRB bit represents the pointed data's multiple reference status. Dedicated instructions *ReadTagWordMrbor* and *Deref* support this operation. *ReadTagWordMrbor* loads memory data pointed by address register into destination register, accumulates both the address register's MRB and the destination register's MRB that is MRB of the memory data, sets the result status in the destination register. *Deref* is similar to the *ReadTagWordMrbor* instruction, but loads memory data into address register and the old address register value is saved to destination register simultaneously. Therefor, succeeding instructions can examine that the pointed data can be reclaimed or not by testing destination register's MRB bit.

These dedicated instructions can minimize the overhead to adopt MRB incremental garbage collection.

3 Memory Architecture

3.1 Cache and bus protocols

Each PIM/p element processing has two 64K bytes caches for instructions and data. PIM/p uses copyback cache protocols which have been proved effective for reducing common bus traffic in shared-memory multiprocessors. To maintain cache coherence, there are basically two mechanisms, invalidating the modified block and broadcasting the new data to others.

PIM/p uses the invalidation method for the following reasons. To use incremental garbage collection MRB, a reclaimed memory area need not be shared. Next time the area is used it may not be shared with the same processors which previously shared the area. In other, KL1 load distribution is achieved by distributing goal records in a cluster from one processor to another. Usually the distributed goals will not be referred from the source processor. In these cases, the broadcast method will produce unnecessary write commands to the common bus on every write to the newly allocated area or distributed goals. The invalidation method is much more efficient.

PIM/p cache protocol is similar to Illinois protocol. However, PIM/p protocol has the following cache commands optimized for KL1. In normal write operations, a fetch-on-write strategy is used; however, it is not necessary to fetch the contents of shared memory when the block is allocated for a new data structure. That means the old data in the block is completely unnecessary. In KL1, when free lists are recreated after grand garbage collection, the old contents of memory have no meanings. To accomplish this, *Direct_Write* is used.

Direct_write: If cache misses at the block boundary, write data into cache without fetching data from memory.

The following instructions are used for inter-processor communication through a shared memory, for example goal distribution.

Read_Invalidate: When cache misses, fetch the block and invalidate the cache block on other CPUs. This operation guarantees that the block is exclusive unless the other CPU subsequently request the block.

Read_Purge: After the CPU reads a block, it is simply discarded even if it is modified.

Exclusive_read: Same as Read_Invalidate except for the last word in a cache block. When it is used to read the last word in a cache block, it purges the block like *Read_Purge*.

Using these instructions, unnecessary swap-in and swap-out can be avoided by invalidating the sender's cache block after receiver the gets the block, and by purging the receiver's cache block after the receiver reads all data in the block.

Ill-behaved software may cause these instruction to destroy cache coherency. However, these instructions are used only in KL1 processing system, and only systems programmers use them.

There are hardware switches which can change the actions of those special read/write instructions to normal read/write actions. By using these switches, the systems programmer can examine their programs consistency.

3.2 Exclusive control operation

To build a shared-memory parallel processor system, lock and unlock operation are essential guarding critical sections. KL1 requires fine-grain parallel processing. The frequency of locking and unlocking operation needed for shared data is estimated at more than 5% of all memory accesses. Thus these operation must be executed

with low overheads by using hardware support. However, locking operations should seldom conflict with each other. It is therefore useful to introduce a hardware lock mechanism which has low overhead when there are no lock conflicts. In PIM/p, the cache block has *exclusive* and *shared* status. When the block is exclusive, it is not owned by other PEs. Hence there is no need to use the common bus. A marker called the lock address register which remembers the block is locked by the CPU. When the CPU locks a block, other CPU cannot get the block data until the block is unlocked by the original CPU. Even when the block is shared, fetching data and invalidating the block before locking is sufficient. The cost is nearly equivalent to the normal write operation.

In KL1 processing, unification requires frequent locking, but the locking time is fairly short. A hardware busy wait scheme is better for lock conflict resolution. If a longer locking time is needed, a software lock can be made by combining *lock*, *read* and *conditional jump* instructions. For KL1, no bus cycles are needed for most of the lock reads hitting exclusive cache blocks.

4 Network Architecture

4.1 Network interface unit

Multiple clusters are connected by a hypercube topology network. At the design stage, we assumed that ten logical reductions require a hundred-bytes packet transfer. The target speed of PIM/p PE will be between 200K LIPS to 500K LIPS. This means 2M to 5M bytes per second network bandwidth is required by each PE. Thus 16M to 40M bytes per second network bandwidth is required to a cluster which contains eight PEs. If this data flows into the common bus, network packet data occupies about 10% to 25% of the total bandwidth of the common bus. Providing a network interface to each processing element reduces such common bus traffic.

Each cluster has 8 PEs, and each PE has a network interface co-processor called a *network interface unit* (NIU). By attaching a NIU to each PE, a PE can send to or receive from a packet without using the common bus. The NIU performs the following functions:

- Builds a packet into the NIU's packet memory, and sends it to the network router(RTR).
- Receives a packet from the RTR, stores it to the packet memory. and signals the arrival of a packet to IPU.
- Communicates to a SCSI bus driver chip which connects to PIM/p front-end processors(FEPs) or disks.

All these actions are controlled by the IPU's co-processor instructions.

To build a packet, the IPU first makes a header which contains the packet destination and mode for broadcasting. It then building a packet body by executing co-processor write instructions, which packs data one, two, or four bytes at a time. Finally the IPU puts a end of packet marker to send the packet to RTR. A whole packet of data is stored in packet memory before sending it, to minimize RTR busy time. The send and receive packet memories are both 16K bytes long.

Each cluster has four SCSI ports which are connected to the PEs. Two have non-differential SCSI interface ports, and the other two have differential SCSI interface ports. The differential SCSI interface is able to extend the interface cable up to twenty five meters. It is used to connect SCSI disks which need not be placed beside the cluster. The PIM/p FEP is connected to a non-differential interface, and various other SCSI devices, such as an ether-net transceiver, can be connected through the SCSI bus. This extends PIM/p's application domain.

4.2 Inter-cluster network connection

While the NIU sends and receives packets, the network packet router(RTR) actually delivers packets. Each RTR connects four NIUs and up to six other RTRs to build a sixth order hypercube network topology. Thus each cluster has two RTRs which construct two independent hypercube networks to improve the total network throughput. The RTR can connect a maximum of sixty-four clusters(512 PEs).

RTR uses the wormhole routing method to reduce traveling time when the network is not so busy, to avoid packet length restrictions caused by RTR packet buffer limitation. Between RTRs data is transferred at system clock rate. RTR has approximately 1K bytes of packet buffer for every output port, in order to reduce network congestion. The static routing method is used and deadlocks are avoided by the routing method. Broadcasting to the sub-cube is available. This can be used when the system is at the initial program stage.

In the PIM/p system, one chassis contains four clusters. The maximum 512PE PIM/p system is sixteen chassis. Building for such a large system can be problematic. Transferring data between these chassis by synchronous-phase matched clock is impossible, because the system occupies an area of about sixteen meters square. This means that the traveling time of data is about one system clock tick. Introducing another hierarchy between inner-chassis communication and inter-chassis communication complicates the distribution strategies of the KL1 processing systems. This should be avoided.

One of main feature of RTR is the interconnection between PIM/p chassis. To attain a transfer rate equal to system clock rate for both inner-chassis and inter-chassis data, RTR uses a data synchronization mechanism for

inter-chassis connections. This makes the inter-chassis connection transfer rate equal to the inner-chassis transfer rate, with little increase in data traveling time. This simplifies the cluster hierarchy.

5 The Kl1 Language Processing System for PIM/p

The KL1 language processing system for PIM/p is designed on the basis of the VPIM [Hirata et al. 1992]; it is the common specifications of the KL1 language processing system on the two level hierarchical multi-processor system. Most specifications of VPIM are used for PIM/p with no changes. Some modification, however, were applied to exploit the PIM/p hardware efficiently.

The KL1 language processing system is implemented as the KL1 compiler and the run-time support routines. The KL1 program must be compiled into PIM/p native machine code when it is executed on PIM/p. The KL1 compiler for PIM/p consists of three passes — the compiler to the intermediate code, the native machine code generator and the optimizer. Compiled KL1 programs may call some run-time support routines as circumstances demand. The run-time support routines are classified into three groups, which correspond to PIM/p memory architecture.

5.1 Changes for PIM/p

There are some changes from VPIM to PIM/p. These were applied to exploit the PIM/p hardware efficiently.

(1) Data Structure

The basic KL1 data are realized by tagged words; each of them consists of a 8-bit tag part and a 32-bit value part, and all KL1 data are realized by tagged words in VPIM. The memory of PIM/p consists of 64-bit width words. Tagged words are placed in aligned 64-bit width words in the PIM/p memory system [Goto et al. 1988]. Although KL1 data density will be low in this scheme, this will not cause performance degradation.

The PIM/p instruction processing unit can access the memory not only in the unit of tagged data, but also in the 8-bit, 16-bit, 32-bit and 64-bit units. A string — an array of integers can, therefore, be realized using 64-bit width words, as shown in figure 6. A *module* which holds KL1 compiled code, is also realized under the same scheme. Since PIMOS [Chikayama et al. 1988] uses many string data and *module* data, this scheme can promote efficiency of memory using.

(2) Data Type Checking

The PIM/p instruction processing unit has special instructions for data type checking: *JumpXorUnderMask* and *JumpNotXorUnderMask*. These have the following functions:

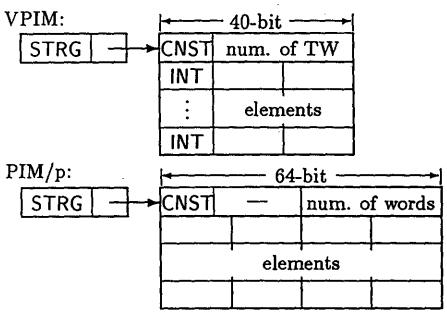


Figure 6: String data of VPIM and PIM/p

$$\text{if(tag\_of(Reg)\&Mask} = \text{Const) goto Label;}$$
and
$$\text{if(tag\_of(Reg)\&Mask} \neq \text{Const) goto Label;}$$

These functions can test not only if the data type is correctly specified, but also if the data type group is correctly specified, since the bit assignment of tag field is designed effectively.

The KL1 language processing system uses 44 kinds of data types; these can be expressed in 6 bits. The tag part, however, is 7-bit width except MRB. We use 7 bits in a tag part to express data type; data types are assigned sparsely in order to check data type group easily by *JumpXorUnderMask* or *JumpNotXorUnderMask*. There are the following data type groups:

- Atomic — atom or integer.
- Vector — null vector, short vector or long vector.
- Short Vector — vector containing 1–8 elements.
- Undefined — variable in some conditions.

These data type groups are often checked in KL1 execution, and this assignment can reduce execution costs.

5.2 Compiler

The KL1 program must be compiled into PIM/p native machine code when it is executed on PIM/p. The KL1 compiler for PIM/p consists of three passes — the compiler to the intermediate code, the native machine code generator and the optimizer. In the first pass, the KL1 program is compiled into intermediate code; its instruction set is called KL1-B. The native machine code generator expands intermediate code into PIM/p native machine code. The optimizer improve the expanded code.

5.2.1 Intermediate Code

In the first pass of the KL1 compiler, the KL1 program is compiled into intermediate code; its instruction set is called KL1-B. It is designed as the instruction set for the *abstract KL1 machine* [Kimura and Chikayama 1987] and interfaces between the KL1 language and the PIM hardware, just as the Warren Abstract Machine [Warren 1983] does for Prolog. The KL1-B for PIM is extended from KL1-B for Multi-PSI to exploit the PIM hardware efficiently.

KL1-B contains passive unification instructions, active unification instructions, argument/element preparation instructions, incremental garbage collection instructions and goal manipulation instructions. These specifications are identical with VPIM [Hirata et al. 1992].

5.2.2 Native Machine Code Generator

The intermediate code, which consists of KL1-B instructions, is expanded into native machine code according to the *template*; the *template* is a set of rules governing translation from KL1-B instructions to native machine instructions. These rules are defined according to the following principles:

- Use the special instructions for KL1 effectively.
- Don't jump in the main pass.
- Minimize the pipeline break ratio.
- Maximize the hit ratio of the instruction cache.

The translating rules are classified into the following 3 groups according to the properties of the KL1-B instructions.

(1) Expand to In-Line Code

These KL1-B instructions which can always be realized by a few native machine instructions are translated accordingly. Consider the following examples:

```
load  Rgp,Pos,Reg
  → ReadTagWordShortOffset   Reg,Pos*8+40(Rgp)
read  Rsp,Pos,Reg
  → ReadTagWordMrbOr         Reg,Pos*8(Rsp)
is_vector  Reg,Lab
  → JumpNotXorUnderMask      Reg,VG,Lab,VGM
put_integer  Const,Reg
  → MoveTagWordWithTag       Rzero,Reg,INT
      (Const = 0)
  → AddImmediate             Reg,Rzero,Const
    MoveTagWordWithTag       Reg,Reg,INT
      (0 < Const < 256)
  → LoadImmediate            Reg,Const
    MoveTagWordWithTag       Reg,Reg,INT
      (Const ≥ 256 or Const < 0)
```

Load is translated into a single native machine instruction. In this sample, *Pos*, the position specifying an argument, is adjusted to the offset in a byte unit.

Read is not a simple read instruction; it must maintain the MRB. PIM/p, however, has a special instruction for this use. *Read* can be realized by a single native machine instruction: *ReadTagWordMrbOr*.

Is_vector tests if the data type group of *Reg* is a vector group. This is translated into a single native machine instruction: *JumpNotXorUnderMask*.

Put_integer has three translation rules from which is selected according to the value of *Const*, in order to generate fast, concise code. These translated codes take 1clock-cycle/4bytes, 2clock-cycles/8bytes and 2clock-cycles/10bytes respectively.

(2) Expand to Conditional Subroutine Call

The KL1-B instructions whose main pass can be realized by a few native machine instructions are translated into these instructions, together with the instructions calling a subroutine conditionally. The subroutines are classified into two groups; the *macro library* and the *roundabout routines*.

The *macro library* is a set of the run-time support routines and called by the *macro call* instruction. These routines realize common functions in executing KL1, and are shared with all compiled codes (See section 5.3). Consider the following examples:

```
reuse_vector  Arity,Reg
  → MacroCallAnd    Reg,MRB,Arity,m_AllocVector
```

Reuse_vector does nothing when the MRB of the vector pointer on the register is not marked. It can, therefore, be translated into a single *conditional macro call* instruction. When the MRB of the pointer is marked, *reuse_vector* allocates a new vector; this allocation is done in the *macro library*.

The *roundabout routine* is placed in the compiled code of the KL1 program. It realizes a local function, and is used from the compiled code of a single KL1-B instruction. Consider the following example:

```
reuse_vector_with_elements  3,Reg,{1,0,1}
  → JumpAnd                  Reg,MRB,LC001
  LR001:
        ⋮
  LC001:
    MacroCall          Rwork1,0,Arity,m_AllocVector
    ReadTagWordMrbOr   Rwork2,0(Reg)
    WriteTagWordShortOffset  Rwork2,0(Rwork1)
    ReadTagWordMrbOr   Rwork2,16(Reg)
    WriteTagWordShortOffset  Rwork2,16(Rwork1)
    JumpDelayed        LR001
    MoveTagWord        Rwork1,Reg
```

Reuse_vector_with_elements is translated into a single *JumpAnd* instruction as a main pass, and some additional instructions as the *roundabout routine*. In KL1 applications, the MRB of the structure pointer is often unmarked, and *roundabout routine* is not executed. This *roundabout routine* is changeable according to the third arguments of the KL1-B instruction. It cannot, therefore, be shared with some KL1-B instructions.

(3) Expand to Subroutine Call

The KL1-B instructions which always execute complicated functions are translated into the subroutine call instruction or the *macro call* instruction. The processing of complicated functions are executed by run-time support routines. Most KL1-B instructions for active unification and body built-in predicates are translated using this rule. This is because the calling cost is low compared to the cost of executing complicated functions, and the size of the compiled code can be minimized.

5.2.3 Optimizer

The compiler for PIM/p supports the optimization of the expanded code; the expanded code is the native machine code translated from the intermediate code according to the *template*. Since expansion according to the *template* is applied to each KL1-B instruction separately, some redundant instructions may be generated, and the order of instructions is not refined. Optimization is applied to the expanded instructions as a group, and these instructions are removed. Two optimization techniques are introduced.

(1) Optimization by Data Flow Analysis

The optimizer analyzes data flow among the instructions in the expanded code. It then trims some redundant instructions and merges some instructions into a single instruction; for example:

- The optimizer trims the instruction which puts a datum onto a register, even if the datum is not used later.
- The optimizer generates an instruction which calculates with a constant datum, instead of an instruction which puts the constant onto a register and an instruction which calculates with the datum on the register.

In this optimization, the data flow analysis is applied separately to the tag part and the value part of a datum. This is because the KL1-B always treats a datum as a set of the tag part and the value part, while some native machine instructions disregard the value part.

The sample code is shown as follows; this is the compiled code of a guard built-in predicate: *add(X,1,Y)*.

- intermediate code:

```
put_integer    1, R2
integer_add    R1, R2, R3, fail
```

- native machine code (not optimized):

```
AddImmediate           R2, Rzero, 1
MoveTagWordWithTag     R2, R2, INT
Add                    R3, R1, R2
JumpAnd                CCR, CC_V, fail
```

- native machine code (optimization #1):

```
AddImmediate           R2, Rzero, 1
Add                    R3, R1, R2
JumpAnd                CCR, CC_V, fail
```

- native machine code (optimization #2):

```
AddImmediate           R3, R1, 1
JumpAnd                CCR, CC_V, fail
```

There are two KL1-B instructions, and each of them is expanded into two native machine instructions. In the unoptimized code, the *Add* instruction uses *R2* as the input, but disregards the value part; therefore the *MoveTagWordWithTag* instruction has no effect and can be removed (optimization #1). Additionally, *AddImmediate R2,Rzero,1* and *Add R3,R1,R2* can be merged into a single native machine instruction: *AddImmediate R3,R1,1*. In this sample, optimized code takes 2clock-cycles/10bytes while unoptimized code takes 4clock-cycles/18bytes.

(2) Pipeline Optimize

The processing element for PIM/p uses a four-stage pipeline. In expanded code, the dependencies between instructions which have been expanded from different KL1-B instructions, are disregarded, and delayed branch instructions are not used as often. The optimizer rearranges the order in which instructions are executed, to ensure smooth pipeline processing.

In KL1 execution, pointer operations — pointer readings and address calculations are often done while pointer operations may cause interlocks. This optimization, therefore, is very effective.

5.3 Run-time Support Routines

The run-time support routines are called from the compiled KL1 program in order to execute complicated functions. They are divided into three groups corresponding to PIM/p memory architecture (Figure 7).

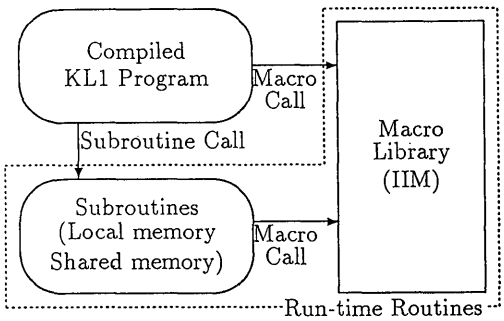


Figure 7: Run-time support routines

(1) Macro Library

The *macro library* is called using *macro call* instructions. This is a kind of subroutine library, but is stored in the *internal instruction memory* (IIM) of IPC, like microprograms. There are no instruction cache misses.

The characteristics of *macro call* instructions are as follows:

- In a *macro call* instruction, a *tag conditional branch*, applied to a run-time KL1 data type check, is carried out in one instruction step.
- Argument registers or short (8-bit) immediate values are specified in the *macro call* instruction, so the operands of a *macro call* can be efficiently passed to its *macro library* function.

The IIM can store 8K-step instructions. We implement the most frequently used functions, for example, the dereference and unification functions, in the *macro library*.

(2) Frequently-used Libraries

Other frequently-used libraries are stored in local memory. The cost of instruction fetches in local memory is small, because it doesn't use the common bus.

Functions for the suspend/resume processes for KL1 goals and the copying GC routines, are implemented in these libraries.

(3) Occasionally-used Libraries

Occasionally-used libraries are stored in shared memory. Access speed for shared memory is slower than that for local memory or IIM, but the storage is so large that we can implement complicated libraries in this memory.

We implement most of the body *built-in* predicates, the network control routines and the *shoen* (meta-function) control routines for these libraries [Hirata et al. 1992].

Table 3: Speedups for Pentomino

Number of PEs	1	2	4	8
Memory shared system	1.00	1.96	3.86	7.50
Network connected system		1.93	3.80	7.28

6 Evaluation

We used Pentomino as a benchmark program and executed it on two system configurations — multi-PE×1CL and 1PE×multi-CL. The multi-PE×1CL configuration represents the memory shared multi-PE system, and the 1PE×multi-CL configuration represents the network connected multi-PE system.

Pentomino is a program to find out all solutions of a 5×8 packing piece puzzle; packing a 5×8 rectangular box by ten various shaped pieces, each is made up of four unit squares. The program does an exhaustive search of an OR-tree of possible pieces elements.

The benchmark program for the network connected multi-PE system contains the *multi-level load balancing* [Furuichi et al. 1990] code which requires the optimization for the network configuration. However, the program for the memory shared multi-PE system does not contains load balancing code.

On the memory shared multi-PE system, the load balancing in a cluster is executed automatically with a KL1 goal as a unit. Each PE has two goal pools, one is local for the PE, the other is public; it is accessible from other PEs. If a PE has many goals in its local pool, it moves some goals into its public pool. The goals in the public pool might be executed by any PEs in the cluster.

Table 3 shows that the speedup ratio according to the number of PEs is nearly equal to the number of PEs for two system configurations. The automatic load balancing mechanism of the memory shared multi-PE system works as efficiently as the optimized load balancing code for the network connected multi-PE system.

7 Conclusion

PIM/p has up to 512 PEs using two level hardware structures. Two level hierarchical structure allows parallel programming with common memory and facilitates system expansion with the hypercube network. On the two level hierarchical structure system, programmers do not think about load balancing inner cluster and write only the load balancing code for clusters.

The special cache control instructions and the macro-call mechanism reduce the common bus traffic, which may become the performance bottle-neck for shared memory multi-PE systems. The evaluation result shows that the speedup is linear upto 8 PEs in a cluster. The com-

mon bus traffic, therefore, does not become the performance bottle-neck.

The macro-call mechanism reduces the costs of type checking and the overhead of passing parameters. Using this mechanism, it becomes easier to implement the KL1 language processing system.

Acknowledgment

We wish to thank all of the PIM research members both at ICOT, at Fujitsu Social Science Laboratory Ltd. and at Fujitsu Limited. Especially we thank ICOT researchers, Dr. K. Hirata and Mr. A. Imai for their useful comments. We also wish to thank Mr. A. Shinagawa and Mr. H. Miyake of Fujitsu Limited for their helpful support in developing softwares. Finally, we would like to thank the director of ICOT research center, Dr. K. Fuchi, the manager of research department, Dr. S. Uchida, the chief of first research laboratory, Dr. K. Taki, the general manager of processor division in Fujitsu Laboratories Ltd, Mr. J. Tanahashi, and the general manager of advanced information systems division in Fujitsu Laboratories Ltd, Mr. H. Hayashi, for their valuable suggestions and guidance.

References

[Asato et al. 1991] A.Asato, M.Kimura, T.Shinogi, A.Hattori. A Pipeline Control Method of PIM/p. In *Proceedings of 43rd Anual convention IPS Japan*, 1991 (In Japanese).

[Chikayama and Kimura 1987] T.Chikayama and Y.Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pp.276–293, 1987.

[Chikayama et al. 1988] T.Chikayama, H.Sato and T.Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230–251, 1988.

[Furuichi et al. 1990] M.Furuichi, K.Taki and N.Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proceedings of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[Goto et al. 1988] A.Goto, M.Sato, K.Nakajima, K.Taki and A.Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208–229, 1988.

[Goto et al. 1990] A.Goto, T Shinogi, T.Chikayama, K.Kumon and A.Hattori. Processor Element Architecture for a Parallel Inference Machine, PIM/p. In *Journal of Information Processing*, pp.174-182, Vol.13, No.2, 1990.

[Hirata et al. 1992] K.Hirata, R.Yamamoto, A.Imai, H.Kawai, K.Hirano, T.Takagi, K.Taki, A.Nakase and K.Rokusawa. Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1992.

[Kimura and Chikayama 1987] Y.Kimura and T.Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pp.468–477, 1987.

[Shinogi et al. 1988] T.Shinogi, K.Kumon, A.Hattori, A.Goto, Y.Kimura and T.Chikayama. Macro-Call Instruction for the Efficient KL1 Implementation on PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.

[Taki 1992] K.Taki. Parallel Inference Machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1992.

[Ueda and Chikayama 1990] K.Ueda and T.Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, (33)6, 1990, pp.494–500.

[Warren 1983] D.H.D.Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Architecture and Implementation of PIM/m

Hiroshi Nakashima*‡ Katsuto Nakajima*
Seiichi Kondo† Yasutaka Takeda*
Yū Inamura† Satoshi Onishi†
Kanae Masuda*

* Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura, Kanagawa 247, Japan

† Institute for New Generation Computer Technology
4-28, Mita 1-Chome, Minato-ku, Tokyo 108, Japan

Abstract

In the FGCS project, we have developed a parallel inference machine, PIM/m, as one of the final products of the project. PIM/m has up to 256 processor elements (PEs) connected by a 16 × 16 mesh network, while its predecessor, Multi-PSI/v2, has 64 PEs. A PE has three custom VLSI chips, one of which is a pipelined microprocessor having special mechanisms for KL1 execution, such as pipelined data typing and dereference.

As for the KL1 implementation on PIM/m, we took much care of garbage collection and introduced various techniques, such as incremental reclamation of local and remote garbage. Especially, a hardware mechanism to support the local garbage collection greatly contributes to reducing the overhead and achieving high peak performance, 615 KLIPS in *append* on single processor.

Sustained performance of single processor is also improved, and is approximately twice as high as that of Multi-PSI/v2. This improvement and the enlargement of the system scale cooperatively enhance the total system performance, and make PIM/m 5 to 10 times as fast as Multi-PSI/v2.

1 Introduction

Several parallel inference machines have been developed in the Japanese Fifth Generation Computer Systems (FGCS) project. As a part of this activity, we have developed three parallel machines. The first machine, Multi-PSI/v1 [Masuda *et al.* 1988, Taki 1988], was an experimental version and was completed in 1986. It has 6 processor elements (PEs) each of which is our first sequential inference machine, PSI-I [Taki *et al.* 1984], and has a software interpreter for the machine language KL1 which is an extended version of flat GHC [Ueda 1985]. Though the machine scale was small and the performance was not very high, the development of Multi-PSI/v1 gave us valuable experimental knowledge of the distributed implementation of KL1 [Ichiyoshi *et al.* 1987].

‡hiroshi@isl.melco.co.jp

The second machine is Multi-PSI/v2 [Takeda *et al.* 1988, Uchida *et al.* 1988], which contains 64 PEs connected by two-dimensional mesh network. Each PE consists of PSI-II's CPU kernel [Nakashima and Nakajima 1987], a network controller, and an 80 MB local memory. KL1 programs are compiled to WAM-like machine instructions for KL1 [Kimura and Chikayama 1987] executed by a microprogrammed emulator. The large machine scale and high performance, owing to the improvement of the processor architecture and implementation technology, make Multi-PSI/v2 the first practical parallel inference machine. Its operating system, PIMOS [Chikayama *et al.* 1988], also greatly contributes to its availability by providing highly sophisticated environment for parallel programming. Thus, many KL1 programs for various application areas have been developed on it since its first model was shipped in 1988 [ICOT 1990]. These programs and many users of 15 systems prove the efficiency and practicality of Multi-PSI/v2.

Then, we have just finished the development of our final machine, PIM/m. It inherits many architectural features, such as the mesh network and KL1 execution mechanism, from Multi-PSI/v2. The performance, however, is greatly improved by drastically modifying PE architecture and increasing the number of PEs to 256.

In this paper, the hardware architecture of PIM/m and the KL1 implementation on it are described. Section 2 shows the system configuration, and the architecture of PE and its processing unit. Section 3 describes several topics about the KL1 implementation emphasizing the relation with garbage collection. Section 4 presents preliminary performance evaluation results and analysis on them.

2 Hardware Architecture

2.1 System Configuration

Figure 1 shows the overview of PIM/m 256 processor system. PIM/m consists of up to 8 cabinets, each of which contains 32 PEs connected to form an 8 × 4 mesh

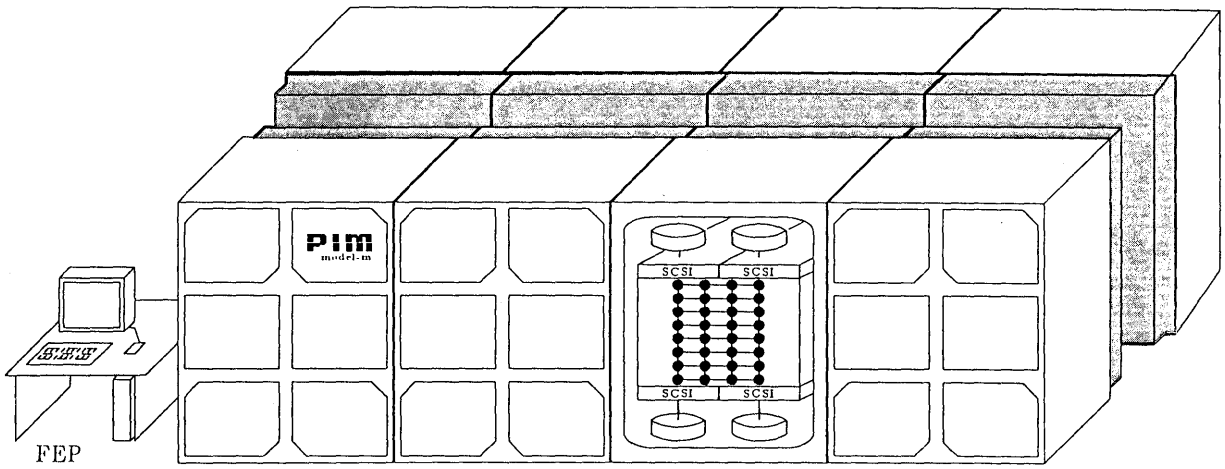


Figure 1: Overview of PIM/m

sub-network. This sub-network is embedded in a larger network, up to 16 × 16, by channels connecting adjacent cabinets. Thus we can provide various size systems, from 32 to 256 PEs.

A cabinet also contains four 670 MB disks, which make a 256 PE system have huge disk space, larger than 20 GB. This huge capacity should be enough for applications such as knowledge base and genetic information analysis. Each disk is coupled with a PE by SCSI bus, which is also used to connect other special I/O devices, other PIM systems, and/or front end processors (FEP).

The FEP is a high performance AI workstation, PSI/UX [Nakashima *et al.* 1990]. It has a special attachment processor to execute a sequential logic programming language ESP [Chikayama 1984]. Since the CPU kernel of FEP is that of PIM/m's PE, FEP is also capable to execute KL1 in single processor environment or simulated multiprocessor environment. Therefore, programmers use FEP not only as an interactive I/O system, but also as a convenient debugging workbench.

2.2 Processor Element

Each PE has three VLSI chips, PU (Processing Unit), CU (Cache Unit) and NU (Network Control Unit), as shown in Figure 2. These chips and other peripheral chips including a floating point processor are installed on one printed circuit board. The other board carries a 16 M-word (80 MB) local memory constructed from 4 M-bit DRAM chips. This two board configuration of PE is much smaller than that in Multi-PSI/v2, eight boards, and makes it possible to increase number of PEs from 64 to 256, owing to the advanced VLSI technology. The machine cycle is 65 ns, which has also been improved from 200 ns of Multi-PSI/v2.

PU is a 40-bit pipelined microprocessor which executes KL1 (and ESP in FEP) under the control of a microprogram stored in 32 K-word writable control store. The architecture of PU is described in 2.3 and 2.4. CU contains a 1 K-word (5 KB) instruction cache and a 4 K-word (20 KB) data cache.

NU performs switching of message packets transferred through the mesh network, using so-called *Worm-Hole Routing* mechanism. As shown in Figure 3, the network of PIM/m consists of full duplex channels connecting adjacent PEs. That is, a pair of adjacent PEs may simultaneously transmit message packets to each other. Moreover, a message packet passing through a PE does not disturb the KL1 execution on the PE, nor collide with others unless they have the same direction.

The network is invested with these properties by the

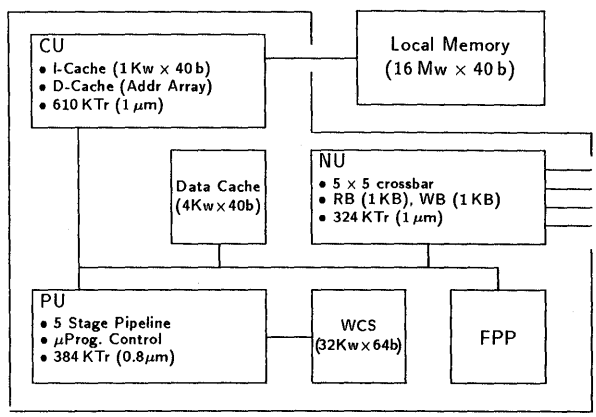


Figure 2: Processor Element

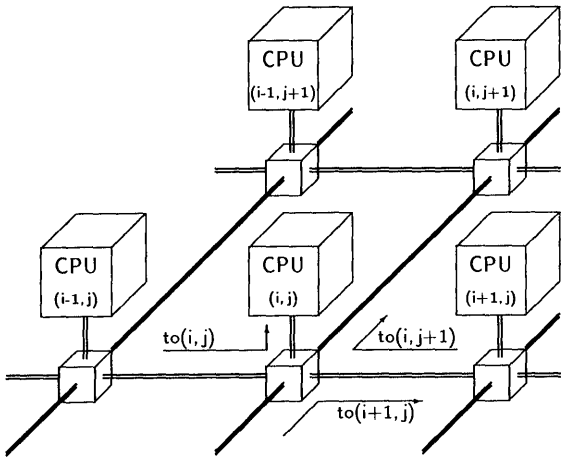


Figure 3: Network Configuration

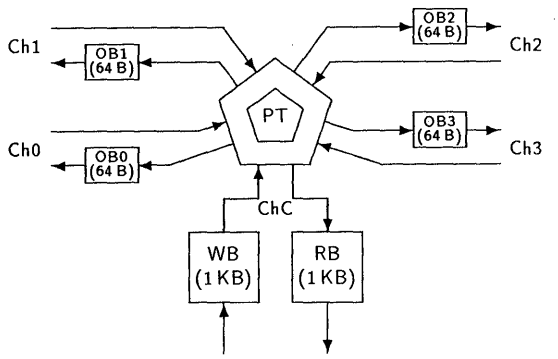


Figure 4: Network Control Unit (NU)

architecture of NU shown in Figure 4. NU has a 5×5 crossbar to switch four input/output channel pairs for adjacent PEs (Ch0–3) and a pair for CPU (ChC). These channels carry a 9-bit packet byte, which consists of 8-bit data field and a *mark* to indicate the header and trailer of a packet.

Switching is performed by looking up a RAM table named *path table* (PT). The address of the table is provided from the packet header which specifies the destination PE number of the packet. Each entry of the table contains a 2-bit code indicating the direction of the packet, going straight, turning left/right, or arriving at its destination. This mechanism gives us much flexibility for routing, system reconfiguration, and physical interconnection of PEs. As the path table has independent read ports for each input channels, collision of packets does not occur even in switching phase.

Once the connection of an input/output channel pair is established, NU transmits a packet byte per four machine cycles regardless the physical location of the adjacent PE. This feature is owing to a sophisticated asynchronous communication mechanism using FIFO *output buffers* (OB0–3). In this mechanism, a sender PE does not wait for acknowledgment from the receiver for a packet byte. Instead it cares about the caution from the receiver saying that the output buffer on the next path will soon be full. Since the caution is raised before the buffer is really full taking physical line delay into account, packet bytes never overrun. The output buffers also contribute to reducing the probability of network choking.

The channel pair for CPU has two FIFO buffers, a *read buffer* (RB) and a *write buffer* (WB). The read buffer acts as an output buffer for the packets directed to the PE itself. Its size 1 KB, however, is much greater than that of output buffers, 64 B, in order to hold a whole message packet. When the tail of a packet written into the read buffer, an interrupt raises to tell CPU that the packet arrives. The write buffer, whose size is also 1 KB, starts transmission of a packet when its tail is written, in order to avoid that the packet is chopped. Both buffers also have the capability to compose/decompose a 40-bit word from/into packet bytes.

2.3 Processing Unit (PU)

Figure 5 shows the configuration of the processing unit, PU [Nakashima *et al.* 1990, Machida *et al.* 1991]. PU executes WAM-like instructions for KL1, named KL1-B [Kimura and Chikayama 1987, Warren 1983], with the following registers.

- A_n/X_n .. Argument and temporary registers.
- PC ..... Program counter.
- AP ..... Alternate clause pointer.
- CGP .... Current goal record pointer.
- GSP .... Goal stack pointer.
- SST ..... Suspension stack top.
- HP ..... Heap pointer.
- SP ..... Structure pointer.
- FVP .... Free variable cell pointer.
- FLP .... Free list cell pointer.
- FGP .... Free goal record pointer.

A_n/X_n are implemented as a register file. The other register file, WR, contains the control registers shown above, except for PC and SP which are hardware counters. Each register is 40 bit width, including 8 bit tag for data type

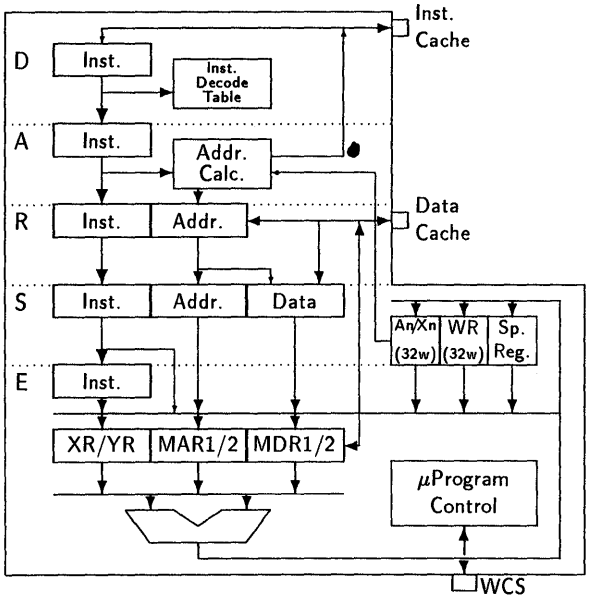


Figure 5: Processing Unit (PU)

representation and incremental/ordinary garbage collection. The tag bit for incremental garbage collection is called *Multiple Reference Bit* (MRB) described in 3.1.

PU has five pipeline stages, **D**, **A**, **R**, **S** and **E**.

The **D** (Decode) stage has a RAM table for instruction decode. Each entry of the table contains the start address of the microprogram routine for an instruction, and the *nano-code* to control the following stages. This RAM decoder makes it easy to develop the microprogram.

The **A** (Address Calculation) stage calculates the operand address by adding two of following resources, according to the nano-code.

- An operand field of the instruction.
- Program counter, PC.
- A_n/X_n specified by an operand field.
- Current goal pointer, CGP, to get a location of a goal argument.

The **A** stage also controls instruction fetch, including conditional and unconditional branch operations.

The **R** (Read Data) stage fetches an operand from data cache using the calculated address, if necessary. The **S** (Setup) stage selects three operands from the following resources and transfers them to the **E** (Execution) stage, according to the nano-code.

- An operand field of the instruction.
- The operand fetched by the **R** stage and its address.
- A_n/X_n specified by an operand field.
- Control registers in WR
- Structure pointer, SP.

In conventional pipelined processors, the operand setup operation is performed by the stage like **R**. PU, however, has an additional special stage, **S**, for the operation. The reason for introducing the **S** stage is that it is required for the pipelined data typing and dereference, as discussed later.

The **E** stage has two pipelined phases controlled by microinstructions. The first phase contains A_n/X_n, WR, and special registers including PC and SP. This phase is shared by the **S** and **E** stages for the operand setup. The second phase has two temporary registers (XR/YR), two memory address registers (MAR1/2), and two memory data registers (MDR1/2). Two of those registers are input to ALU, and the result is written into registers in the first and/or second phase. ALU operation and tag manipulation including turning on/off MRB are performed in parallel.

2.4 Data Typing and Dereference

Data typing and dereference are very important for efficient implementation of logic programming languages. Both data typing and dereference are performed by checking the tag of data and changing the control flow according to the result. PU has powerful mechanisms, including the pipelined data typing and dereference, for these operations.

The **E** stage has the following microprogram operations for tag checking.

(1) Two-way conditional jump. The jump condition is obtained by comparing the tag of a register with an immediate value or the tag of another register.

(2) Three-way jump. The tag of MDR1 or MDR2 is compared with an immediate value and *reference* tag.

(3) Multi-way jump. A RAM table, which contains jump offsets, is looked up by the tag of MDR1 or MDR2.

These operations requires two machine cycles. The first cycle makes the jump condition or offset, and the second generates the jump address and fetches the microinstruction.

The pipelined data typing and dereference, which are most unique features, mainly depend on the **S** stage. The **S** stage has the following three functions for data typing.

(1) Modify the microprogram entry address comparing the tag of the operand fetched by the **R** stage with an immediate value.

(2) Set up the offset of a multi-way jump, which can be performed by the first microinstruction, looking up the RAM table by the tag of the operand fetched by the **R** stage.

(3) Set up the two-way jump condition, which can be examined by the first microinstruction, comparing the tag of an operand transferred to the **E** stage with an immediate value.

The first two functions require the special stage between the **R** and **E** stages.

The **S** stage also performs dereference. When the dereference from A_n/X_n is ordered, the **R** stage fetches the operand if the A_n/X_n contains reference pointer, while it always fetches the operand in the case of the dereference from memory. In both cases, the **S** stage examines the tag of fetched data, and repeatedly reads memory until a non-reference data is obtained. The state of the reference path indicated by MRB of each reference pointer is also examined, as described in 3.1.

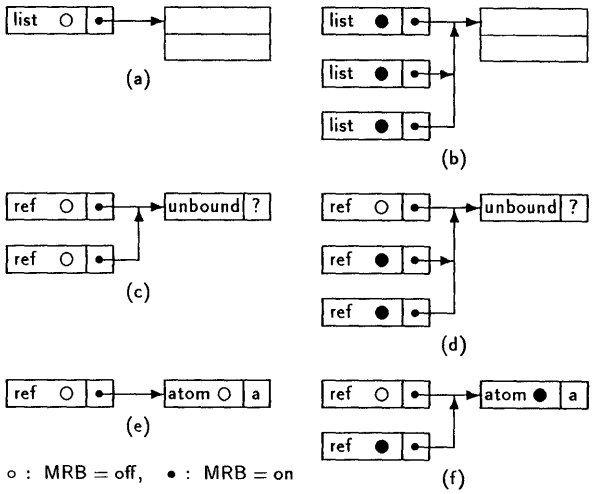


Figure 6: Multiple Reference Bit (MRB)

3 Implementation

Since logic programming languages don't have destructive assignment, manipulating a data structure often makes a copy of the data leaving its old version as a garbage. In Prolog, garbage data cells may be reclaimed by intentional backtrack with side effect operations. In KL1, however, this technique cannot be used because *deep* backtrack causes the failure of the entire program. Thus, garbage reclamation has to be performed only by the run-time system.

In the KL1 implementation on PIM/m, therefore, we took much care of garbage collection and its efficiency. For the reclamation of *local* garbages, an incremental garbage collection using *Multiple Reference Bit* (MRB) is introduced. *Remote* garbages, which was once pointed from PEs other than its home, are also reclaimed incrementally by a sophisticated reference counting mechanism for reducing the number of inter-PE messages, called *Weighted Export Counting* (WEC).

This section describes the implementation of KL1, emphasizing these garbage collection mechanisms and related techniques to reduce memory space and number of messages.

3.1 Local Incremental Garbage Collection

Concurrent processes in KL1 communicate each other through shared logical variables. Typically, a pair of concurrent processes, a producer and a consumer, has its own logical variable in which the producer puts some data by an *active* (or *body*) unification. The consumer will be activated by binding the variable and read its contents by a *passive* (or *guard*) unification. Because the variable cell is shared only by the producer/consumer pair, it will become garbage after the consumer gets its contents. Moreover, a structured data unified with the variable may also become garbage after the consumer decomposes it.

The *Multiple Reference Bit* is introduced in order to reclaim these garbages [Chikayama and Kimura 1987]. MRB is a one-bit reference counter attached to both pointers and objects. As the counter for a pointer, MRB is turned on (overflowed) if the pointer is duplicated, as shown in Figure 6(a) and (b). That is, a pointer with MRB on might refers to an object together with other pointers. In other words, an object directed by a pointer with MRB off can be reclaimed as a garbage after the (passive) unification is performed through the pointer.

This rule, however, has an exception for unbound variables each of which can have two reference pointers with MRB off, for a producer and a consumer (Figure 6(c)). After the producer unifies the variable with some data and loses its reference path to the variable, the path from the consumer to the data is left alone as the rule requires.

This exception leads to the other aspect of MRB, counter for an object. As shown in Figure 6(d), an unbound variable might have a pointer with MRB off and two or more pointers with MRB on. If the variable is unified with a data through the pointer with MRB on, the data has a pointer with MRB off, although it cannot be reclaimed by the unification through the pointer. Thus the data, which is an atomic or a pointer, should have MRB indicating whether it is pointed by multiple

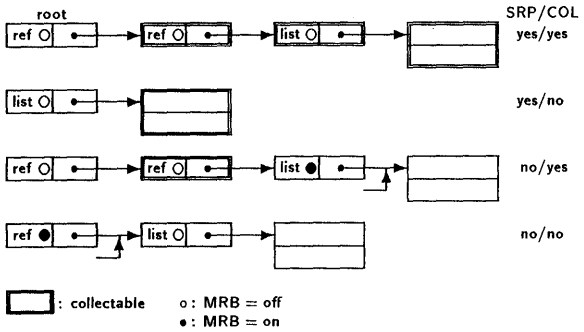


Figure 7: Pipelined Dereference Supporting Incremental Garbage Collection

pointers as shown in Figure 6(e) and (f).

The incremental garbage collection is mainly performed when the unifier makes dereference. A member of the chain of reference pointers can be reclaimed if both its MRB and that of its predecessor are off. The terminal of the chain is also reclaimable if the same condition is satisfied. Especially, all of the members on the chain is collectable if their MRBs are off.

In order to support the reclamation, the pipelined dereference mechanism of PU maintains the following information (Figure 7).

SRP (Single Reference Path):
MRBs of all the pointers on the chain are off.

COL (Collectable):
MRBs of the first two pointers are off.

These are not only passed to the **E** stage, but also combined with the data typing result to make microprogram entry address, in order that the **E** stage easily decide whether the reclamation can be done.

On passive unification, if the dereference result is a structure, the structure will be collected after the commit operation. For example, the instruction "`collect_list`" is located at the beginning of the body code for a clause having head unification with a list cell, and reclaims the list cell if the path to it is single. For the processes filtering streams represented by lists, "`reuse_list`" is used for passing the list cell directly rather than putting and getting it to/from the free cell pool. To these instructions, SRP is passed through the MRBs of their operands, A_n/X_n.

SRP is also examined by built-in predicates for optimization [Inamura *et al.* 1989]. For example, "`set_vector_element`" updates an element of a vector to make its new version, providing the path to the vector is single. The stream merger also examines the state of the paths to the variable cell representing a stream and the list cell to be put, in order to reuse these cells.

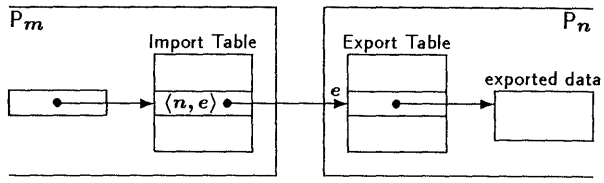


Figure 8: Export Table and Import Table

3.2 Remote Incremental Garbage Collection

As MRB overflows when an object has two or more pointers, there are garbages which cannot be reclaimed by the incremental garbage collection mechanism described in 3.1. Therefore, a PE may exhaust its local memory space and invoke a batch-mode garbage collector. In order to allow the garbage collector to move data around in the local memory space, remote references are indirected with *export table* as shown in Figure 8 [Ichiyoshi *et al.* 1987]. A remote reference consists of the pair of identifiers for PE and the export table entry from which the exported data is pointed. Thus, a PE is free to perform batch-mode garbage collection independently, because other PEs are ignorant of local data addresses but aware of positions of the table entries which never move.

The other indirection table for remote references, *import table* in Figure 8, is introduced to reclaim export table entries incrementally. Entries for single-referenced objects are easily reclaimed using MRB scheme. When a PE, P_e, exports the pointer to a single-referenced object to another PE, P_i, it registers the pointer into *MRB-off* export table. P_i also registers the remote reference into *MRB-off* import table in order to identify that the remote path is single. Unless P_i duplicates the path to the import table entry, the export table entry is reclaimed when P_i makes a remote access to the object. For example, when P_i wants to read the object, it sends a message to get the object. The message also says that the remote path is single, and causes reclamation of the export table entry by P_e. On the other hand, if P_i makes multiple paths and then loses them all, the reclamation is triggered by batch-mode garbage collector on P_i. After the *marking* of the garbage collection, the import table is scanned to find out unmarked entries and send a message for each of these entries for the reclamation of corresponding export table entry.

In order to reclaim export table entries for multiple-referenced objects, we introduced *Weighted Export Counting* (WEC) method [Ichiyoshi *et al.* 1988], which is also independently proposed in [Watson and Watson 1987]. A PE pair, P_e and P_i, exporting and importing the pointer to a multiple-referenced object has entries

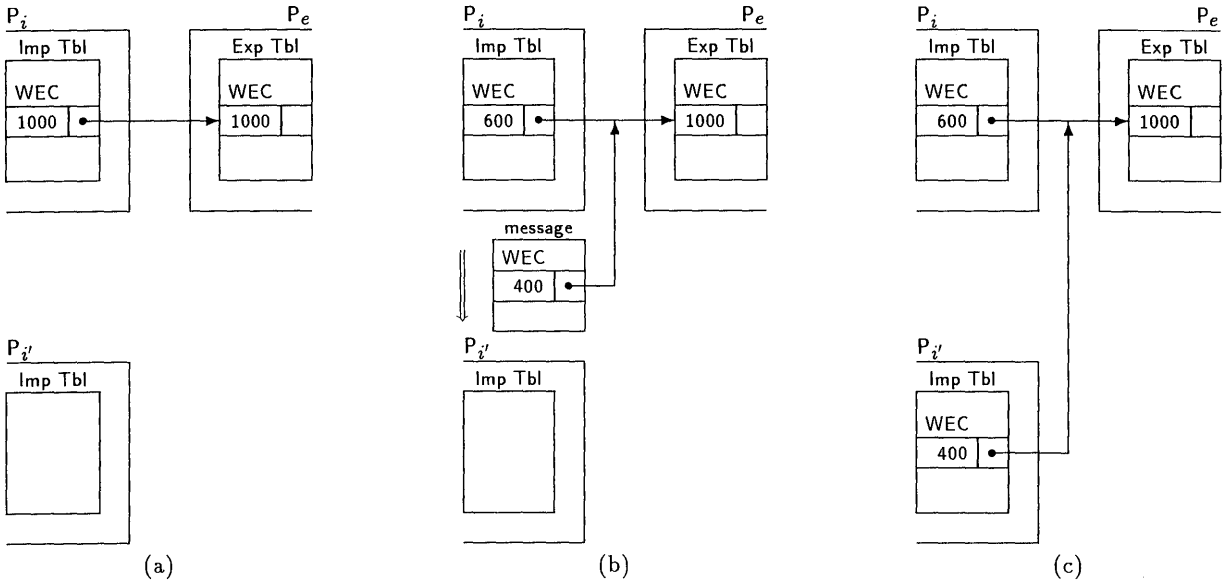


Figure 9: Weighted Export Counting (WEC)

on *MRB-on* export/import table for the object. Each entry has a slot for WEC value, which is a kind of reference counter but is initiated with some large number, say 1000, rather than one, as shown in Figure 9(a). When P_i duplicates the pointer and exports one of the results to another PE, $P_{i'}$, it divides the WEC value into two portions, say 600 and 400. Then P_i sends a message containing the remote reference and the WEC value 400 to $P_{i'}$ (Figure 9(b)). $P_{i'}$ receives the message and makes an import table entry with the WEC value 400 (Figure 9(c)). Note that the total of WEC values associated with the remote references is equal to the WEC value of export table entry through all phases shown in Figure 9. If P_i (or $P_{i'}$) finds that there are no paths to an import table entry on incremental or batch-mode garbage collection, it sends a message for reclamation to P_e with the WEC value. The WEC value of the export table entry is decremented by that in the message, and the entry is reclaimed if the WEC value becomes zero.

This scheme has the advantage of ordinary reference counting, because it omits request and acknowledgment messages which the ordinary scheme requires when P_i exports the pointer to $P_{i'}$. That is, in the ordinary scheme, P_i should send a message for incrementing the reference counter to P_e, and suspend exporting until it receives acknowledgment from P_e. Unless P_i wait for the acknowledgment, the counter on P_e might be cleared transitively by the decrement request from $P_{i'}$ which possibly reaches P_e earlier than the increment request from P_i.

A similar *weighted* counting method, *Weighted Throw Counting* (WTC), is adopted to detect the termination of a group of goals [Rokusawa *et al.* 1988]. KL1 has the capability to supervise goal groups, called *Shōen*, as if they are meta-interpreted [Chikayama *et al.* 1988]. For example, the operating system PIMOS can detect the termination of a user program represented as a Shōen. Since goals in a Shōen may be distributed to many PEs, some remote reference counting is necessary to detect the termination of them all. As WEC for remote references, WTC values are given to PEs executing goals in a Shōen. Thus, PEs can exchange goals with some WTC values omitting requests/acknowledgments as described before. This feature is very important for efficient execution because an active unification with a remote variables is a goal.

3.3 Multiple Export and Import

Once a multiple-referenced object is exported, it is often exported again. If such a object is repeatedly exported overlooking that it has been already exported, each time an export table entry is consumed. A PE importing such a object repeatedly, worse still, gets multiple copies of the object. In order to solve these problems, both export table and import table are content addressable by hashing. The hash table for export associates (local) addresses of exported objects with export table entries, while that for import associates remote references with import table entries.

This scheme, however, cannot deal with more complicated situations. For example, if P_i imports two pointers from P_e and $P_{e'}$, and each pointer refers to a copy of a

Table 1: Single Processor Performance

benchmark	condition	PIM/m	Multi-PSI/v2	$\frac{\text{Multi-PSI/v2}}{\text{PIM/m}}$
append	1,000 elements	1.63 msec	7.80 msec	4.8
best-path	90,000 nodes	142 sec	213 sec	1.5
pentomino	8×5 box	107 sec	240 sec	2.2
15-puzzle	5,885 K nodes	9,283 sec	21,660 sec	2.3

data structure on each PE, P_i will get multiple copies from P_e and $P_{e'}$. This troublesome situation may occur in distribution of program codes which have intricate cross references.

Therefore, we introduced global identification of code modules to promise that a PE should not have multiple copies of a code module [Nakajima *et al.* 1989]. When P_e is requested by P_i to send a data object and find out that the object is a code module, it transmits the module identifier rather than the module itself as the reply. Then P_i looks up a hashed table for modules resident in it with the identifier. If the module is resident, P_i simply executes it. Otherwise, P_i sends a special message for getting the module itself to P_e.

4 Performance Evaluation

4.1 Single Processor Performance

Table 1 shows the single processor performance of PIM/m for four benchmarks. The table also includes the performance of Multi-PSI/v2 and the ratio of PIM/m and Multi-PSI/v2 (M/P-speedup) to show the effect of architectural improvement.

The performance for *append* represents the *peak* performance which is 4.8 times as high as that of Multi-PSI/v2. This improvement should greatly owe to pipelined data typing and dereference, because the speedup factor for major **E** stage operations is only 1.5 (two 65 ns cycle versus one 200 ns cycle). The effectiveness of pipelined dereference supporting the incremental garbage collection is proved by the fact that the speedup factor is significantly larger than 4.2* for Prolog *append* on PSI-II and PSI/UX whose CPU kernels are those for Multi-PSI/v2 and PIM/m respectively [Nakashima *et al.* 1990].

On the other hand, the absolute performance, 615 KLIPS, is still lower than 1.4 MLIPS for Prolog on PSI/UX. A part of this dereference is caused by the fact that the incremental garbage collection mechanism inherently requires additional memory accesses to free cell pool and variables excluded from list cells. In fact, KL1 *append* performs 10 memory accesses per one reduction in our system, while Prolog *append* does 6 accesses required essentially. The other part, however, should be due to the hardware support for the incremental garbage collection which is not yet sufficient to remove the overhead. For example, we estimated that some modifications of the hardware with few gates for;

- making information indicating whether the dereferenced path is totally collectable,
- fetching an element from free variable pool in pipeline, and
- storing the result of an ALU operation into both the structure pointer and an argument register

will make the performance 810 KLIPS.

The other three benchmarks are search programs with various parallel algorithms and load distribution strategies. *Best-path* finds out the shortest path between two vertices of a directed weighted graph with a parallelized Dijkstra's algorithm and static load distribution [Wada-K and Ichiyoshi 1989]. *Pentomino* makes OR-parallel exhaustive search to solve a packing piece puzzle problem with a multiple level dynamic load distribution method [Furuichi *et al.* 1990]. *15-puzzle* solves a well-known puzzle problem in parallel by employing iterative-deepening A* algorithm [Wada-M and Ichiyoshi 1991]. Although these programs are not practical, the algorithms and load distribution strategies should be generally adopted to various application programs of parallel processing. Thus, it is expected that the performance for them reflects the performance sustainedly gotten on PIM/m.

The M/P-speedup for these program, 1.5 to 2.3, are not excellent in contrast with the case of *append*. This is probably caused by two major reasons, context switch and cache miss. In these programs, context switches frequently occur, every two to three reductions, by the termination or suspension of goals, while never in *append*. Since instructions for the context switch take dozens of cycles for execution in the **E** stage and make pipeline stagnant, the pipelined architecture doesn't gain much performance improvement for these programs.

*This value is normalized to compensate the machine cycle difference between Multi-PSI/v2 and PSI-II.

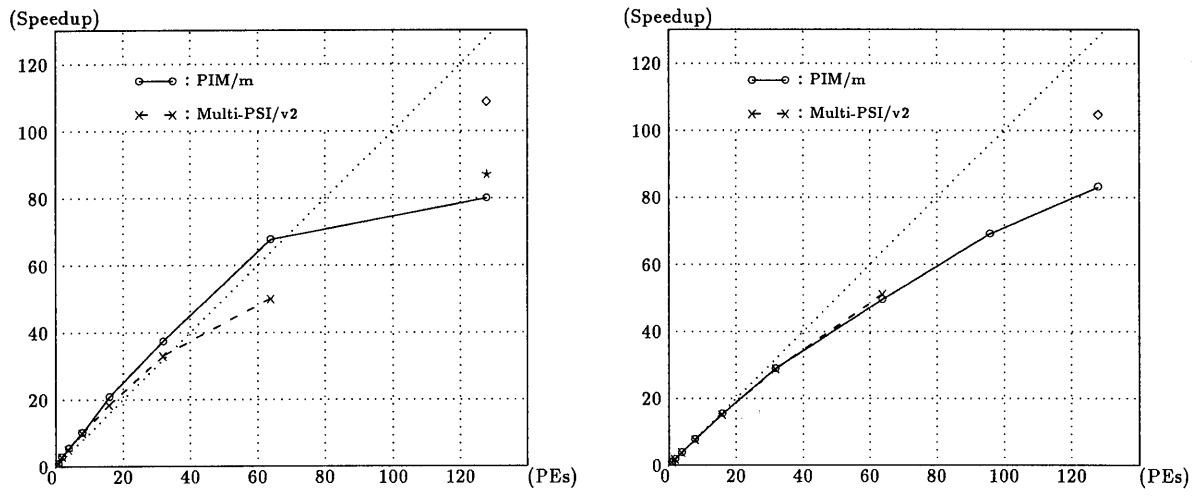


Figure 10: Speedups for *best-path* and *pentomino*

Cache miss penalty should be the major degradation factor in *best-path* which has a large working set. Even in Multi-PSI/v2, cache miss degrades the performance 10 to 20 % as reported in [Nakajima and Ichiyoshi 1990]. Thus, the penalty relative to the machine cycle becomes more critical, because the cache size and physical memory access time of PIM/m are not greatly evolved from Multi-PSI/v2.

4.2 System Performance

System performance is strongly related with load distribution strategy and communication cost. Since PIM/m has four times as many PEs as Multi-PSI/v2 has, it might become difficult to balance loads distributed to PEs. As for communication cost, we evaluated that the network capacity of Multi-PSI/v2 is much larger than required [Nakajima and Ichiyoshi 1990]. Therefore, we designed PIM/m's network making its throughput and bandwidth almost equal to those of Multi-PSI/v2's, expecting that the network still has enough capacity. The frequency of message passing, however, might be contrary to our expectation, because of underestimation of hot spot effect and so on.

The speedup, which is gotten by dividing execution time for single processor by that for n processors, may give preliminary answers about those questions. Figure 10 shows the speedups of PIM/m and Multi-PSI/v2 for *best-path* and *pentomino*. Up to the 64 PE system, the speedup of PIM/m are quite similar to or slightly better than that of Multi-PSI/v2. Especially, the result of *best-path* shows surprising super-linear speedup, probably because partitioning the problem makes required memory space for a PE small and reduces cache miss rate and/or the frequency of batch-mode garbage collection. These results show that the network of PIM/m stands increase of message passing frequency caused by the improvement of PE performance. Thus, the perfomance of single cabinet minimum system is greatly improved from Multi-PSI/v2. That is, M/P-speedup is 5.6 for *best-path* and is 8.3 for *pentomino*.

On the other hand, the speedup of the 128 PE system are considerably low, especially for *best-path*. Thus, the M/P-speedups for 4-cabinet a half of maximum system are 3.7 for *best-path* and 6.4 for *pentomino*. This implies that the problem size is too small to distribute loads to 128 PEs and/or the message passing frequency exceeds the network capacity. As for *best-path*, the reason of low speedup seems to be small size of the problem which takes only 1.8 sec on the 128 PE system, because a PE transmits messages only to its adjacent PEs. For example, when the problem is scaled up by increasing the number of nodes from 90 K to 250 K and 1 M, the speedups for the 128 PE system become 87 and 109 respectively, as shown in the figure*.

*Since *large* problems cannot run on small size systems, the speedups are estimated by multiplying 32 PE speedups for small problems by 32 to 128 PE speedups for large problems.

In *pentomino*, its load distribution strategy might cause hot spot PEs which pool loads and distribute them in demand driven manner. The hot spot, however, is possibly that of computation for load generation rather than communication for distribution. The problem size may also limits the speedup, because the execution time of the 128 PE system is only 1.3 sec. The speedup of larger size problem, which is for 10×6 box and takes 211 sec on the 128 PE system, is 105 as shown in the figure*. We are now planning further evaluation and analysis to confirm these observations or find out other reasons.

As for *15-puzzle*, we measured the speedups of 64 and 128 PE systems changing the problem size as shown in Figure 11. The figure also shows the number of nodes in the search space for each of seven initial states of the game board. The results for the 64 PE system of PIM/m is also quite similar to that of Multi-PSI/v2. The speedup of the 128 PE system, 38.7 to 109.2, are tightly related to the size of problems. The analysis of this relation is also left as a future work.

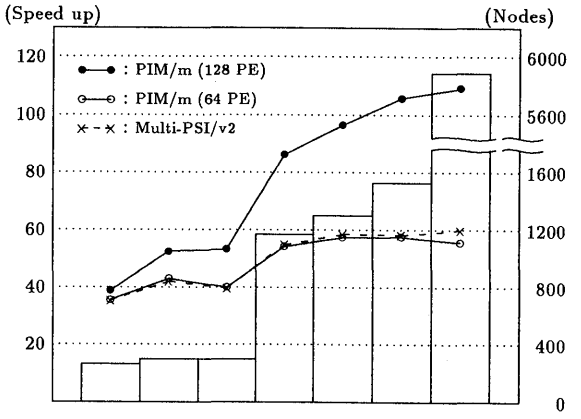


Figure 11: Speedup for *15-puzzle*

5 Concluding Remarks

This paper presented the hardware architecture of PIM/m system, its processor element, and the pipelined microprocessor dedicated to the fast execution of KL1 programs. The KL1 implementation issues focused on its relation with garbage collection were also described. Then preliminary performance evaluation results were shown with brief discussions on them.

We are now planning a research concentrated on further evaluation of the performance of PIM/m and the behavior of various KL1 programs. The evaluation results and detailed analysis on them should greatly contribute not only to the performance tune-up of PIM/m but also to the research on parallel inference machines in next step.

Acknowledgment

We would like to thank all those who contributed to the development of PIM/m system in ICOT, Mitsubishi Electric Corp. and related companies. We also wish to thank Vu Phan and Jose Uemura for their contribution to this paper.

References

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 292–298, Nov. 1984.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. 4th Intl. Conf. on Logic Programming*, pp. 276–293, 1987.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 230–251, 1988.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki, and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50–59, Mar. 1990.

[Ichiyoshi *et al.* 1987] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proc. 4th Intl. Conf. on Logic Programming*, pp. 257–275, 1987.

[Ichiyoshi *et al.* 1988] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 904–913, Nov. 1988.

[ICOT 1990] ICOT. *Proc. Workshop on Concurrent Programming and Parallel Processing*, 1990.

[Inamura *et al.* 1989] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Technique Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. North American Conf. on Logic Programming 1989*, pp. 907–921, 1989.

[Kimura and Chikayama 1987] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proc. 4th IEEE Symp. on Logic Programming*, pp. 468–477, Sept. 1987.

[Machida *et al.* 1991] H. Machida, H. Andou, C. Ikenaga, H. Nakashima, A. Maeda, and M. Nakaya. A 1.5 MLIPS 40-

bit AI Processor. In *Proc. Custom Integrated Circuits Conf.*, pp. 15.3.1–15.3.4, May 1991.

[Masuda *et al.* 1988] K. Masuda, H. Ishizuka, H. Iwayama, K. Taki, and E. Sugino. Preliminary Evaluation of the Connection Network for the Multi-PSI system. In *Proc. 8th European Conf. on Artificial Intelligence*, pp. 18–23, 1988.

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, 1989.

[Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-Processor Communication in the KL1 Implementation on the Multi-PSI. In *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 613–614, Aug. 1990.

[Nakashima and Nakajima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proc. 4th IEEE Symp. on Logic Programming*, pp. 104–113, Sept. 1987.

[Nakashima *et al.* 1990] H. Nakashima, Y. Takeda, K. Nakajima, H. Andou, and K. Furutani. A Pipelined Microprocessor for Logic Programming Languages. In *Proc. 1990 Intl. Conf. on Computer Design*, pp. 355–359, Sept. 1990.

[Rokusawa *et al.* 1988] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 18–22, Aug. 1988.

[Takeda *et al.* 1988] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama, and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 978–986, Sept. 1988.

[Taki *et al.* 1984] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 398–409, Nov. 1984.

[Taki 1988] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI System. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. North-Holland, 1988.

[Uchida *et al.* 1988] S. Uchida, K. Taki, K. Nakajima, A. Goto, and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 16–36, Nov. 1988.

[Ueda 1985] K. Ueda. Guarded Horn Clauses. Technical Report 103, ICOT, 1985. (Also in *Concurrent Prolog: Collected Papers*, The MIT Press, 1987).

[Wada-K and Ichiyoshi 1989] K. Wada and N. Ichiyoshi. A Study of Mapping of Locally Message Exchanging Algorithms on a Loosely-Coupled Multiprocessor. Technical Report 587, ICOT, 1989.

[Wada-M and Ichiyoshi 1991] M. Wada and N. Ichiyoshi. A Parallel Iterative-Deepening A* and its Evaluation. In *Proc. KL1 Programming Workshop '91*, pp. 68–74, May 1991. (In Japanese).

[Warren 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, Oct. 1983.

[Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In *Proc. Parallel Architecture and Languages Europe*, June 1987.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1

Keiji HIRATA Reki YAMAMOTO Akira IMAI Hideo KAWAI
Kiyoshi HIRANO Tsuneyoshi TAKAGI Kazuo TAKI

Institute for New Generation Computer Technology
4-28 Mita 1 chome, Minato-ku, Tokyo 108 JAPAN

Akihiko NAKASE
TOSHIBA Corporation

Kazuaki ROKUSAWA
OKI Electric Industry Co.,Ltd.

Abstract

This paper focuses on a parallel and distributed implementation method for a concurrent logic programming language, KL1, on a parallel inference machine, PIM. The KL1 language processor is systematically designed and implemented. First, the language specification of KL1 is deliberately analyzed and properly decomposed. As a result, the language functions are categorized into unification, inter-cluster processing, memory management, goal scheduling, meta control facilities, and an intermediate instruction set. Next, the algorithms and program modules for realizing the decomposed requirements are developed by considering the features of PIM architecture on which the algorithms work. The features of PIM architecture include a loosely-coupled network with messages possibly overtaken, and a cluster structure, i.e. a shared-memory multiprocessor portion. Lastly, the program modules are combined to construct the language processor. For each implementation issue, the design and implementation methods are discussed, with proper assumptions given.

This paper concentrates on several implementation issues that have been the subjects of intense ICOT research since 1988.

1 Introduction

In the Fifth Generation Computer Systems Project, ICOT has been, simultaneously, developing a large-scale parallel machine PIM [Goto *et al.* 1988] [Imai *et al.* 1991], designing a concurrent logic programming language KL1 [Ueda and Chikayama 1990], and investigating the efficient parallel implementation of KL1 on PIM [ICOT 1st Res. Lab. 1991]. These subjects are closely related and have been evolving together.

The KL1 language has several good features: a declarative description, simple representation of synchronization and communication, symbol manipulation, parallelism control, and portability. Similarly, PIM architecture, also, has a number of good features: high scalablity, general purpose applicability, and efficient symbolic computing.

When implementing KL1 on PIM, various difficulties appear. However, the parallel and distributed implementation of KL1 must bridge the semantic gap between PIM and KL1 so that programmers can enjoy the KL1 language as an interface for general-purpose concurrent/parallel processing [Taki 1992].

ICOT has implemented KL1 on Multi-PSI (a distributed-memory MIMD machine) and has been accumulating experience in KL1 implementation [Nakajima *et al.* 1989]. The implementation of KL1 on Multi-PSI was a preliminary experiment for our implementation.

This paper primarily focuses on a parallel and distributed implementation method for the concurrent logic programming language KL1 on a parallel inference machine PIM. Section 2 gives readers some brief background knowledge on PIM and KL1. Section 3 systematically investigates the complex connections of what part of the language specification is supported by what component(s) of the KL1 language processor. Among these components, Section 4 focuses on and discusses several key implementation issues: efficient parallel implementation within a shared-memory portion, inter-cluster processing, a parallel copying garbage collector, meta control facilities, and a KL1 compiler. Section 5 concludes this paper.

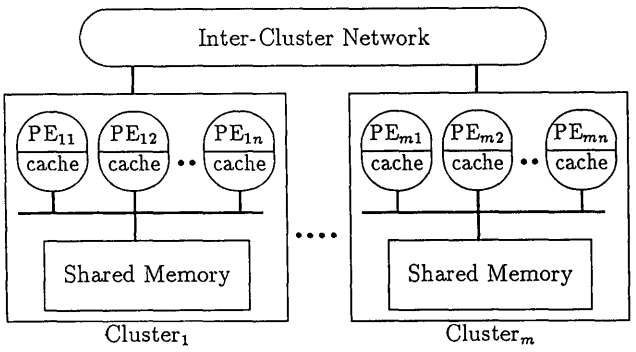


Figure 1: PIM Architecture

2 Overviews of PIM and KL1

2.1 PIM

Figure 1 shows the PIM architecture [Goto *et al.* 1988] [Imai *et al.* 1991]. PIM architecture assumptions and features are described below.

One of the features of PIM architecture is its hierarchy. Up to about ten processing elements (PEs) are interconnected by a single bus to form a structure called a "cluster" in which main memory is shared. Here, the bus can be regarded as a local network. Many clusters can be interconnected by a global network. Within a cluster, inter-PE communication can be realized by short-delay high-throughput data transfer via the bus and the shared memory. Thus, PEs within a cluster share their address spaces, and each PE has its own snooping cache. The instruction set of a PE includes `lock&read`, `write&unlock`, and `unlock` as basic memory operations.

Inter-cluster communication, though, may pass messages through some relay nodes and over long distances. Thus, inter-cluster communication increases the time delay and decreases the throughput. The address spaces of distinct clusters are separated, of course. The network delivers message packets to destinations while reading their header and tailer information.

PIM architecture assumes the following property for the inter-cluster loosely-coupled network. If PEs send and/or other PEs receive message packets, the order of packets does not obey the FIFO rule. Even in one-PE-to-one-PE communication, the FIFO rule is not obeyed. This assumption comes from the following hardware characteristics of PIM architecture. The reasons for this assumption are as follows. One is that there may be more than one path between two clusters [1]. The other is that when more than one PE within a cluster simultaneously sends message packets, it is not determined that which packet will be launched first into the network. In this sense, in the loosely-coupled network of PIM, messages

[1]However, the routing of the PIM network is not adaptive.

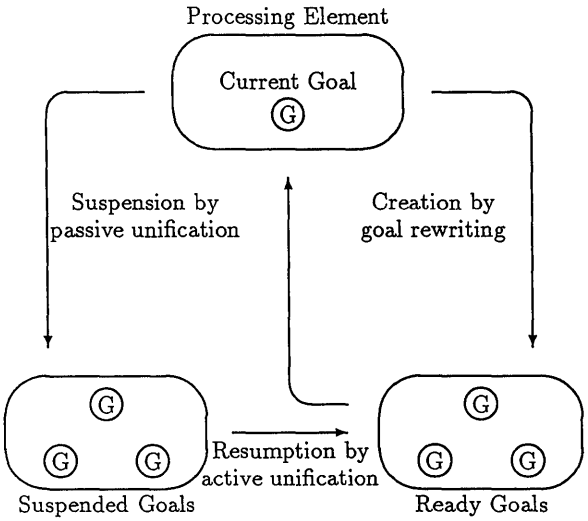


Figure 2: KL1 Execution Image

are possibly *overtaken* in the network.

2.2 KL1

KL1 is a kernel language for the PIM based on the GHC (Guarded Horn Clauses) language [Ueda and Chikayama 1990]. Figure 2 shows our KL1 execution image. A clause of a KL1 program can be viewed as a rewrite rule, which rewrites to the body goals a goal that succeeds the guard unification and satisfies the condition (guard), and has a form as follows:

$$\underbrace{p : -g_1, ..., g_m}_{guard\ part} \mid \underbrace{q_1, ..., q_n}_{body\ part}.$$

Where p, g_i, and q_i stand for predicates. This rewriting of a goal is also called *reduction*. The execution model has a goal pool which holds the goals to be rewritten. Goals are regarded as lightweight processes. Basically, guard goals $g_1, ..., g_m$ and body goals are reduced concurrently, thus yielding parallelism.

Goal (process) communication is realized as follows. Suppose that more than one goal shares a variable. When a goal binds a value to the shared variable, a clause for rewriting the other goal that shares the variable may be determined. The value which is instantiated to the shared variable controls the clause selection; this is the communication between KL1 goals.

Synchronization is realized as follows. When a goal is going to determine which clause can be used for rewriting, and the variables included in the goal are uninstantiated, the unification and the guard execution may be deferred since there is not enough information for the clause selection. The uninstantiated variables are supposed to be shared and the other goal is expected to bind

a value to the variable afterwards. Consequently, the suspended goal reduction waits for variable binding for the clause selection. That is, variable instantiation realizes data-flow synchronization. Actually, the KL1 language processor must deal efficiently with frequent suspension and resumption.

Even if more than one clause can be used for rewriting, just one clause is selected indeterminately. A vertical bar between the guard part and the body part '|', called a commit operator, designates indeterminacy. Since it is sufficient to hold a single environment for each variable, efficient implementation is expected.

One of features of the KL1 language is the provision of simple yet powerful meta control facilities as follows: goal execution control, computation resource management, and exception handling. These are essential for designing efficient parallel algorithms and enabling flexible parallel programming. Usually, operating systems perform meta-control on a process basis. However, the KL1 language aims at fine-grain parallelism, and the KL1 language processor reduces a large number of goals in parallel. Therefore, it is inefficient and impossible for a programmer or the runtime system [2] to control the execution of each goal. Consequently, KL1 introduces the concept of a shoen [3] [Chikayama *et al.* 1988]. A shoen is regarded as a goal group or a task with meta-control facilities. An initial goal is given as an argument to the built-in predicate shoen; descendant goals belonging to the shoen are controlled as a whole. Descendant goals inherit the shoen of the parent goal. Shoens are possibly nested as well; the structure connecting shoens is a tree.

Moreover, to realize sophisticated mapping of parallel computation, *priority* and *location specification* are introduced; that is, they can be used for programming speculative computation and load balancing. If a programmer attaches an annotation to a body goal e.g. `p@priority(N)`, this tells the runtime system to execute goal p at priority `N`. Moreover, a goal can have a location specification e.g. `p@cluster(M)`; this designates the runtime system to execute the goal p in the `M` th cluster. These two specifications are called *pragmas*. These pragmas never change the correctness of a program although they change the performance drastically.

3 Systematic Design of KL1 Language Processor

When implementing KL1 on PIM, various kinds of difficulties appear. Firstly, although the PIM architecture adopts a hierarchical configuration, the KL1 implementation has to provide a uniform view of the machine to programmers. Secondly, it is difficult to determine to what extent a runtime system should support the functions of KL1 and which functions it should support within the specification of KL1. For instance, since the KL1 language does not specify the goal-scheduling strategy, a runtime system can employ any scheduling algorithm. However, both the general-purpose and the efficient algorithm are generally difficult to develop. Thirdly, for efficient implementation, it is important to employ algorithms which include fewer bottlenecks in terms of parallel execution. Lastly, the KL1 language processor is complex and of a large scale.

Therefore, it is a promising idea to be able to overcome these difficulties by systematically designing a language processor as follows. Firstly, the given language specification must be deliberately analyzed and properly decomposed. Then, the algorithms and the program modules for realizing the decomposed requirements must be developed by considering the machine architecture on which the algorithms work. Lastly, the designer must construct the language processor by combining the program modules. A good combination of these modules will yield an efficient implementation. We designed the KL1 language processor on a loosely-coupled shared-memory multiprocessor system (PIM) by following these guidelines.

3.1 Requirements

At first, we summarize the required functions of the KL1 language processor into the four items in the leftmost column of Table 1. These items are the result of analysis and decomposition of the KL1 language specification. The KL1 language processor may look like the kernel of an operating system.

Next, mechanisms which satisfy these requirements are divided into those supported by a compiler and those supported by a runtime system. Furthermore, mechanisms by the runtime system are divided into two levels according to the machine configuration of PIM: shared-memory level and distributed-memory level (the topmost row of Table 1).

Some of the technologies used for KL1 implementation on single-processor systems may be expanded to shared-memory multiprocessor systems. That is because both systems suppose a linear memory address space. However, it may not be straightforward to expand the single-processor technologies to distributed-memory multiprocessor systems in general. Of cource, that is mainly because distributed-memory systems provide a non-linear memory address space. Thus, the techniques used for distributed-memory systems are possibly quite different from those for a single-processor system.

The contents of Table 1 show our solutions; that is,

[2] The software modules of the KL1 language processor executed at run time are called a runtime system as a whole. For instance, the runtime system may include an interpreter, firmware in microcode, and libraries. On the contrary, compilers, assemblers and optimizers are not included in a runtime system.

[3] Shoen is pronounced, '*show*' '*N*'.

Table 1: Implementation Issues of this Paper

	Compiler	Runtime System	
		Shared-memory Level	Distributed-memory Level
Unification	Decomposition	Suspension and Resumption	Message Protocol
Memory Management	Reuse inst.	Local GC	Export and Import Tables Weighted Export Count
Goal Scheduling	TRO	Automatic Load Balancing	
Meta-control			
Execution Control		Termination Detection	Foster-parent Weighted Throw Count Message Protocol
Resource Management		Resource Caching	
Exception Handling			

what techniques are used for parallel and distributed KL1 implementation. Each item in the leftmost column of the table is mentioned below.

3.1.1 Unification

Goals are distributed all over a system for load balancing and may share data (variables and ground data) for communication. Logical variables remain resident at their original location. Consequently, not only intra-cluster but also inter-cluster data-references appear. During unification, goals have to read and write the shared data consistently and independently from the timings and locations of goals and data. Thus, mechanisms for preserving data consistency are needed.

As described above, goals are rewritten in parallel and, thus, variable instantiations occur independently from each other. Suspension and resumption mechanisms based on variable bindings control goal execution and realize data-flow synchronization.

Hence, our KL1 implementation must realize the mechanisms for data consistency, synchronization, and unification in a parallel and distributed environment. Moreover, since a major portion of the CPU time is spent for unification, the algorithm should be concerned with efficiency.

3.1.2 Memory Management

Logical variables inherently have the single-assignment property. The single assignment property is very useful to programmers, but gives rise to heavy memory consumption. Since the KL1 language does not backtrack, KL1 cannot perform memory reclamation during execution as Prolog does. Thus, an efficient memory management mechanism is indispensable for the KL1 language processor. The issues associated with memory management are allocation, reclamation, working-set size, and garbage collection. To achieve high efficiency, not only must the algorithms and the data structure of the runtime system be improved, but also a compiler has to generate effective codes by predicting the dynamic behavior of a user program as much as possible.

3.1.3 Goal Scheduling

The KL1 language defines goal execution as concurrent. Thus, the system is responsible for the exploitation of actual parallelism. One implementation issue associated with goal scheduling is determining which goal scheduling strategies have high data locality, yet keep the number of idle PEs to a minimum.

Further, the KL1 language provides the concept of goal priority; each KL1 goal has its own priority as explicitly designated by a programmer. Then, goals with higher priorities are *likely* to be reduced first. Goal prioritization in KL1 is weak in some respect. Under the goal priority restriction, it is crucial to achieve load balancing.

3.1.4 Meta Control Facilities

The goals of a shoen may actually be distributed over any clusters, and, thus, goals may be reduced on any PE in the system. Since the system operates in parallel, shoens are loosely managed; it is simply guaranteed that each operation will finish eventually. That is, it is impossible to execute a command simultaneously to all the goals of a shoen.

A shoen has two streams as arguments of the `shoen` built-in predicate; one is for controlling shoen execution, and the other is for reporting the information inside the shoen. A shoen communicates with outside KL1 processes through these two streams. Messages, such as `start`, `stop`, and `add_resource`, enter the control stream from the outside. Messages, such as `terminated`, `resource_low`, and `exception` return to the report stream from the inside.

It is very difficult to evaluate the CPU time and memory space spent for computation when goals are distributed and executed in parallel. Therefore, the current system regards the number of reductions as a measure of the computing resources consumed within the shoen.

The exceptions reported from a shoen include illegal input data, unification failure[4], and perpetual suspension.

Some examples of shoen functions are shown below.

`Stop` message: When a `stop` message is issued in the control stream of a shoen, the system has to check whether or not the goals to be reduced belong to the shoen, and, if they do, the shoen changes its status to `stop` as soon as possible. The `stop` message is propagated to the nested descendant shoens.

Resource Observation: The system always watches the consumption of computation resources, that is, the total number of times goals belonging to each shoen are reduced over the entire system. If the amount of consumption within a shoen is going to exceed the initial amount of supplied resources, the system stops the reduction of shoen goals and, then, issues the `resource_low` message on the report stream, viz. a supply request for a new resource.

Exception Handling: When a programmer or the system creates an exception during the reduction of a goal in a shoen, the shoen responsible recognizes the exception and converts the exception information to a report stream message [5]. The exception of the KL1 language is concerned with illegal arguments, arithmetic, failure, perpetual suspension and debugging. An exception message on the report stream indicates which goal caused what exception and where. Additionally, the exception message includes variables for a continuation given from the outside; the other process can designate a substitute goal to be executed, instead of the goal causing the exception.

3.2 Overview of Implementation Techniques

ICOT developed the Multi-PSI system in 1988 [Nakajima *et al.* 1989]. The KL1 system is running on the Multi-PSI. The architecture of PIM is very different from that of Multi-PSI in the following two points. One is that PIM has a loosely-coupled network with messages possibly overtaken. The other is that PIM has cluster structures that are shared-memory multiprocessors. Due to these features, PIM attains high performance, and, at the same time, the complexity of the KL1 language processor increases.

This section describes many of the implementation techniques we have been developing for such an architecture. Among these techniques, the issues which this paper focuses on are listed in Table 1.

[4]Notice that the unification failure of a KL1 goal does not influence the outside of a shoen. In this sense, the reduction of a KL1 goal never fails, unlike GHC.

[5]The mechanism for creating and recognizing exceptions is similar to catch-and-throw in LISP.

3.2.1 Unification

The synchronization and communication of KL1 are realized by read/write operations to variables and suspension/resumption of goal reduction during unification. These operations are described below.

Passive Unification and Suspension: Passive unification is unification issued in the guard part of KL1 programs. The KL1 language does not allow instantiation of variables in its guard part. The guard part unification is nonatomic. Since KL1 is a single-assignment language, once a variable is instantiated, the value never changes. This means that passive unification is simply the reading and comparing of two values. From the implementational point of view, basically only read operations to variables are performed. Thus, no mutual exclusion is needed in the guard part.

If goal reduction during the guard part is suspended, the goal is hooked to variables. Here, we have an assumption that almost all goals wait for a single variable to be instantiated afterwards. Therefore, an optimization may be taken into account; the operation for the goal suspension is just to link the goal to the original variable. If multiple uninstantiated variables suspend goal reduction, however, the goal is linked to the variables through a special structure for multiple suspension. During passive unification, only these suspension operations modify variables; the operations are realized by the compare & swap primitive.

Active Unification and Resumption: Active unification is unification issued in the body part of KL1 programs. The KL1 variables are allowed to be instantiated only in the body part. When an instantiation of a shared variable occurs, if goals are already hooked to the variable, these goals have to be resumed as well as the value assignment. When instantiating a variable, since other PEs might be instantiating the variable simultaneously, mutual exclusion is required. We also adopt compare & swap as the mutual exclusion primitive.

When unifying two variables, one variable has to be linked to another to make the two variables identical. At this time, other PEs might be unifying the same two variables. Therefore, imprudent unification operation might turn out to generate a loop structure and/or dangling references. To avoid these, the following linking rule should be obeyed: the variable with the lowest address is linked to the one with the highest.

Section 4.1 describes the implementation of unification in detail.

3.2.2 Inter-cluster Processing

In a KL1 multi-cluster system, more than one PE in each cluster reduces goals in parallel. If a goal reduction succeeds, there are two kinds of new goal destination: the cluster that the parent goal belongs to and the other cluster. If the other cluster is designated for load balancing, the runtime system *throws* the new goals to the clusters. If the arguments of a goal to be thrown are references to variables and structures, the references across clusters consequently appear, these are called *external references.* Here, suppose that a new goal with reference to data in cluster A is thrown to cluster B. Then, original cluster A *exports* the reference to the data to cluster B, and foreign cluster B *imports* the reference to the data from cluster A. Exportation and importation are also implemented by message sending. Multiple reference across clusters inevitably occurs.

An external reference is straightforwardly represented by using the pair <*cl, addr*> where *cl* is the cluster number in which the exported data resides, and *addr* is the memory address of the exported data. This representation of an external reference provides programmers with a linear memory space.

However, this implementation causes a crucial problem; efficient local garbage collection is impossible. Here, *local* means that garbage collection is performed locally within a cluster. See Section 4.3 for more details on garbage collection. Since our local garbage collector adopts a stop and copy algorithm (Section 4.3), the locations of data move after garbage collection. At that time, all of the new addresses of moved data should be announced to all other clusters. Thus, straightforward representation would make cluster-local garbage collection very inefficient.

Section 4.2 shows our solution to this problem and discusses more detailed inter-cluster processing subjects.

3.2.3 Memory Management

As described in Section 3.1.2, the implementation of memory management should pay close attention to allocation, reclamation, working set size, and garbage collection.

Allocation and Reclamation: A cluster has a set of free lists for pages and supports any number of contiguous pages [6]. These are called global free lists. The size of pages is uniform; supposedly the integral power of two [7]. A PE has a set of free lists for data objects, the sizes of which are less than the page size. These are called private free lists. Actual object size is rounded up to the closest integral power of two; the private free lists just support the quantum sizes of 2^n. Moreover, objects contained in a page are uniform in size.

A PE allocates an object as follows. When a PE requires an object which is smaller than a page, the PE first tries to take an object from an appropriate private free list. If a PE runs out of a private free list and fails to take an object, then the PE tries to take a new page from the global free lists. If it succeeds, the PE partitions the page area into objects of the size the PE requires, recovers the starved free list and, then, uses an object. Otherwise, if a PE cannot take a proper page area from a global free list, the PE tries to extend the heap to allocate a new page area on demand. When a PE requires an object which is larger than a page, the PE tries to take new contiguous pages from global free lists. Otherwise, the PE tries to extend a heap to allocate new contiguous pages as above.

When a PE reclaims a large or small object, it is linked to the proper free list.

The features of this scheme are as follows:

- Since a PE has its own private free lists for small objects, the access contention to global free lists and the heap is alleviated.
- A PE usually just links garbage objects to and takes new objects from appropriate free lists; it leads the small runtime overhead for allocation and reclamation [8].
- Since every PE handles its private free lists using push and pop operations (obeying the LIFO rule), the working set size can be kept small.
- Since the size of small objects is rounded up to the nearest 2^n, the number of private free lists to be managed decreases, and the deviation of private free list lengths can be alleviated to some extent. Additionally, the fragmentation within a page is prevented, though some objects might contain unused areas.
- Since this scheme does not join two contiguous objects, unlike the buddy system, its runtime overhead of reclamation is kept small.

On the other hand, when the free list of some size run out, our KL1 language processor does not partition a large object into smaller ones, but allocates a new page. This is mainly because, due to too much partitioning, it is likely that garbage collection will be invoked even if only slightly large object is required. The other reasons are as follows. In general, it is inefficient to incrementally partition a small object into even smaller objects. The overhead for searching an object to be partitioned is needed. Also, in our KL1 language processor, a local stop-and-copy garbage collector (described just below, (2)) collects garbages and rearranges the heap area efficiently.

[6] Currently, there are 15 kinds of free lists for supported pages: $1 \sim 15 - and - more$.

[7] The size of a page is currently 256 words.

[8] A module of PIM, PIM/p, has dedicated machine instructions for handling free lists, ***push*** and ***pop***.

Furthermore, a KL1 compiler optimizes memory management by generating codes not only for allocation and reclamation but also to reuse data structures utilizing the MRB scheme [Chikayama and Kimura 1987] (Section 4.6.4).

Garbage Collection: Our KL1 language processor performs three kinds of garbage collections

(1) local real-time garbage collection using the MRB scheme
(2) local stop-and-copy garbage collector
(3) real-time garbage collection of distributed data structures across clusters.

Since (1) can reclaim almost any garbage object, (2) is needed, eventually. (1) has a very small overhead and can defer the invocation of (2). Moreover, in a shared-memory multiprocessor, it is important that (1) does not destroy data on snooping caches and keeps the working set size of an application program small [Nishida *et al.* 1990], unlike (2). Section 4.3 discusses the parallel copying garbage collector (2) in detail. Section 4.2.2 discusses our method for reclaiming data structures referred to by external reference (3) in detail.

3.2.4 Goal Scheduling

The aim of goal scheduling is to finish the execution of application programs earlier. It is impossible for a programmer to schedule all goals strictly during execution. In particular, in the knowledge processing field, there are many programs in which the dynamic behavior is difficult to predict. The optimum goal scheduling depends on applications, and, thus, there are no general-purpose goal scheduling algorithms. Hence, a programmer cannot avoid leaving part of the goal scheduling to a runtime system. Then, PEs within a cluster share their address spaces, and the communication between them is realized with a relatively low overhead. Optimistically thinking, the performance will pay for the overhead of the automated goal-scheduling within a cluster as the number of PEs increases. However, when the automated goal-scheduling for inter-cluster does not work well, the penalty is even greater. Consequently, the KL1 language processor adopts automated goal-scheduling performed within a cluster and manual goal-scheduling among clusters.

Furthermore, the runtime system should schedule goals fairly by managing priorities. Section 4.4 discusses the implementation of goal scheduling.

3.2.5 Meta Control Facilities

The meta control facilities of KL1 are provided by a shoen. The implementation model for a shoen on a distributed environment introduces a *foster-parent* to prevent bottlenecks and to realize less communication. A

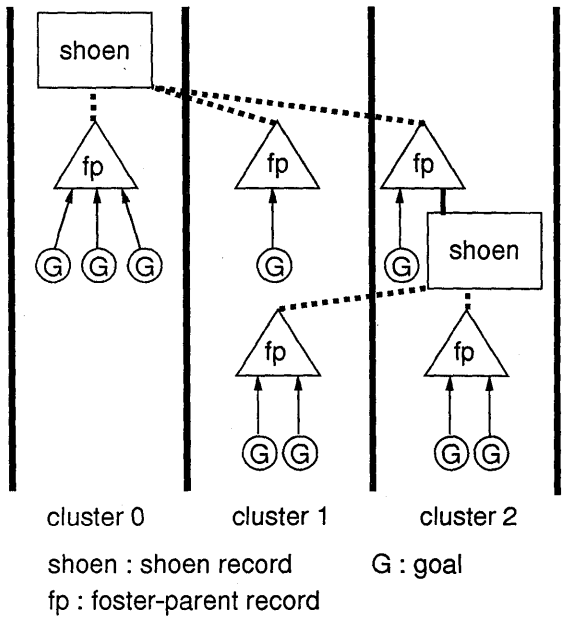


Figure 3: Relationship of Shoen and Foster-parents

foster-parent is a kind of proxy shoen or a branch of a shoen; the foster-parents of a shoen are located on clusters where the goals of the shoen are reduced.

A shoen and a foster-parent are realized by record structures which store their details, such as status, resources, and number of goals. Figure 3 shows the relationship between shoens, foster-parents and goals.

As in Figure 3, a shoen controls its goals and the descendant shoens resident in a cluster through a foster-parent of the cluster. A shoen directly manages its foster-parents only. Then, a foster-parent manages the descendant shoens and goals.

A shoen is created by the invocation of the **shoen** predicate. At that time, a shoen record is allocated in the cluster to which the PE executing the **shoen** predicate belongs. Next, when a goal arrives at a cluster but the foster-parent of its shoen does not yet exist, a foster-parent is created for the goal execution automatically. During execution, new goals and new descendant shoens are repeatedly created and terminated. When all goals and descendant shoens belonging to a foster-parent are terminated, the foster-parent is terminated, too. Further, when all foster-parents belonging to a shoen are terminated, the shoen is terminated.

On comparing a shoen record and a foster-parent record of our implementation with those of the Multi-PSI system, ours must hold more information because of the PIM network with messages possibly overtaken. That is, in our KL1 system, the automatons to control a shoen and a foster-parent require more transition states.

Consequently, in terms of implementing a shoen and a foster-parent, we have to pay special attention to efficient protocols between a shoen and its foster-parents

which work on the loosely-coupled network of PIM (messages are possibly overtaken in the PIM). Another point requiring attention is that, since parallel accessing might become a bottleneck, the system should be designed so that such data do not appear, i.e. less access contention. Section 4.5 describes the parallel implementation of a shoen and a foster-parent in more detail.

3.2.6 Intermediate Instruction Set

As described so far, the KL1 language processor is too large and complex to be implemented directly in hardware or firmware. To overcome this problem, we adopted a method suggested by Prolog's Warren Abstract Machine (WAM) [Warren 1983] where the functions of the KL1 language processor are performed via an intermediate language, KL1-B. The advantages of introduction of an intermediate language include: code optimization, ease of system design and modification, and high portabilty.

The optimization achieved at the WAM level brings about more benefits than the peep-hole optimization since the intermediate instruction sequence reflects the meanings of the source Prolog program. Similarly, the optimization at the KL1-B level gains more than the peep-hole optimization. Details on the optimization are described in Sections 4.6.4 and 4.6.5.

If the specification of the KL1-B instruction set is fixed, it is possible to independently develop a compiler for compiling KL1 into KL1-B and a runtime system executing the KL1-B instructions. If a runtime system can be designed so that it absorbs the differences in hardware architecture, the machine-dependent parts of the KL1 language processor are made clear, and portability is improved.

3.2.7 Built-in Predicates

This section mentions the optimization techniques on the implementation of the built-in predicates `merge` and `set_vector_element`. These techniques were originally invented for the Multi-PSI system. Our KL1 language processor basically inherits the techniques.

merge: The `merger` predicate merges more than one stream into another. It is useful for representing indeterminacy; actually, the `merge` predicate is invoked frequently in practical KL1 programs, such as the PIMOS operating system [Chikayama *et al.* 1988]. Although a program for a stream merger can be written in KL1, the delay is large. Thus, it is profitable to implement the merger function with a constant delay by introducing the `merge` built-in predicate.

Let us consider a part of a KL1 program:

```
.., p(X), q(Y), merge(X,Y,Z), ..
```

When predicate `p` is to unify `X` and its output value, a system merger is invoked automatically within the unifier of `X`. The same thing happens as `Y` of `q`. See [Inamura *et al.* 1988] for a more detailed discussion.

set_vector_element: To write efficient algorithms without disturbing the single-assignment property of logical variables, the primitive can be used as follows in the KL1 language:

```
set_vector_element(Vect, Index, Elem,
                   NewElem, NewVect)
```

When an array `Vect`, its index value `Index`, and a new element value `NewElem` are given, this predicate binds `Elem` to the value at the position of `Index` and `NewVect` to a new array which is the same as `Vect` except that the element at `Index` is substituted for `NewElem`. Using the MRB scheme, our KL1 language processor detects a situation that `NewVect` is obtained in constant time. That is, the situation is that the reference to `Vect` is single, and, thus, destructive updating of the array is allowed. See [Inamura *et al.* 1988] for a more detailed discussion.

4 Implementation Issues

This section focuses on several important implementation issues which ICOT has been working on intensively for the past four years.

Our implementation mainly takes the following into account:

- *Smaller and shorter mutual exclusion within a cluster*
  If the locking operation is effective over a wide area or for a long time, system performance is seriously degraded due to serialization. To avoid this, scattered and distributed data structures are designed, and only the *compare & swap* operation is adopted as a low-level primitive for light mutual exclusion [9].
- *Less communication; i.e., fewer messages*
  Since inter-cluster communication costs more than inner-cluster communication, mechanism for eliminating redundant messages are effective.
- *Main path optimized while enduring low efficiency in rare cases*
  Since the efficiency of rare cases does not affect total performance, the implementation for handling the rare cases is simplified and low efficiency is endured. This is important for reducing code size.

Important hardware restrictions to be taken into account are:

[9] Higher-level software locks contain this primitive.

- *Snooping caches within a cluster; data locality has a great effect*
  It is important to keep the working set of each PE size small. This leads to a reduction in the shared bus traffic and increase in the hit ratio of the snooping caches.
- *Messages are possibly overtaken in the loosely-coupled network of PIM*
  The number of shoen states and foster-parent states to be maintained increases. The message protocol between clusters should be carefully designed.

4.1 Unification

The unification of variables shared by goals realizes synchronization and communication among goals. Since more than one PE within a cluster performs unification in parallel, mutual exclusion is required when writing a value to a variable.

Since unification is a basic operation of the KL1 system, efficiency greatly affects total performance. At first, this section shows simple and efficient implementation methods of unification. Next, since problems associated with the loosely-coupled network of PIM occur, a distributed unification algorithm which works consistently and efficiently on the network is presented.

4.1.1 Simplification Methods

There are two ways to simplify the unification algorithm as follows.

Structure Decomposition: A KL1 compiler decomposes the unification of a clause head. For example, (a) of the following program is decomposed to (b) at compile time.

```
p([f(X)|L]) :- true          | q(X), p(L).   (a)
p(A) :- A = [Y|L], Y = f(X) | q(X), p(L).   (b)
```

Thus, the compiler can generate more efficient KL1-B code corresponding to (b).

Substitution for System Goals: In rare cases, a runtime system automatically substitutes part of the unification process with special KL1 goals. This can alleviate the complexity of a unification algorithm; implementors need not pay attention to mutual exclusion of the part. For example, let us consider the following two rare cases.

- A compare & swap failure (another PE has modified the value); If this happens, then the following KL1 goal is automatically created and scheduled as if it were defined by a user:

```
unify_retry(X,Y) :- true | X = Y.
```

  The above `X` and `Y` are unified to variables one at least of which has failed compare & swap during unification.

- Active unification of two structures is invoked; All elements of the two structures should be unified, however, the operation is rather complex (the ordinal implementation uses stacks like Prolog). To simplify the operation for rare cases, a special KL1 goal is ordinarily created and scheduled. For example, if two active unification arguments are both lists, the following goal is created.

```
list_unifier([X1|X2], [Y1|Y2]) :- true |
      X1 = Y1, X2 = Y2.
```

4.1.2 Distributed Implementation Based on Message Passing

The principle of the protocol for distributed unification is as follows. A read/write operation to an external reference cell (Section 4.2.1) basically causes a corresponding request message to be launched to the network. However, redundant messages are eliminated as much as possible.

Distributed Passive Unification: Passive unification has two phases: reading and comparing. First, to execute the read operation on an external reference cell is to send a `read` message to the foreign exported data. If the exported data has become a ground term (an instantiated variable), an `answer_value` message returns. If the exported data is still a variable, the request message is hooked to the variable. If the data is an external reference cell, the `read` message is forwarded to the cluster to which the cell refers.

Next, the `answer_value` message arrives at the original cluster. Then, the returned value is assigned to the external reference cell, and the goal waiting for the reply message is resumed. Eventually, the goal reduction is going to compare the two values. Moreover, the import table entry for the cell can be released.

The efficient implementation of inter-cluster message passing itself is presented in Section 4.2.

Safe and Unsafe Attributes: If an argument of active unification is an external reference cell, the active unification has to realize the assignment in a remote cluster. Sending a `unify` message to the exported data assigns a value to the original exported data. However, in general, the unification of two variables from distinct clusters may generate a reference loop across clusters. In order to avoid creating such reference loop, we introduce the concept of *safe/unsafe external references* [Ichiyoshi *et al.* 1988]. When there is active unification between a variable and an external reference cell, and the external reference cell is *safe*, it is possible that the variable

is bound to the external reference cell. If the external reference cell is *unsafe*, a `unify` message is sent to the exported data.

4.2 Inter-cluster processing

4.2.1 Export and Import Tables

Export Table: As described in Section 3.2.2 , straightforward implementation of an external reference makes cluster-local garbage collection very inefficient.

In order to overcome this problem, each cluster introduces an *export table* to register all locations of data which are referenced from other clusters (Figure 4). That is, exported data should be accessed indirectly via the export table. Thus, the external reference is represented by the pair <*cl, ent*>, called the *external reference ID*, where *ent* is the entry number of the export table. As the export table is located in the area which is not moved by local garbage collection, the external reference ID is not affected by local garbage collection. Changes in the location of exported data modify only the contents of export table entries.

Since exported data is identified by its external reference ID, distinct external reference IDs are regarded as distinct data even if they are identical. To eliminate redundant inter-cluster messages, exported data should not have more than one external reference ID. Thus, every time a system exports an external reference ID, the system has to check whether or not the external reference ID is already registered on the export table.

Import Table: In order to decrease inter-cluster traffic, the same exported data should be accessed as few times as possible. Hence, each cluster maintains an *import table* to register all imported external reference IDs. The same external references in a cluster are gathered into the same internal references of an *external reference cell* (EX in Figure 4).

Then, exported data is accessed indirectly via the external reference cell, the import table, and the export table.

The external reference cell is introduced so that it can be regarded equally as a variable. Operations to a variable are substituted for the operations to the external reference cell.

Every time the system imports an external reference ID, the system has to check whether or not the external reference ID is already registered in the import table. Thus, the import table entry and the external reference cell point to each other.

4.2.2 Reclamation of Table Entries

As described above, the export table is located in the area which is not moved by local garbage collection.

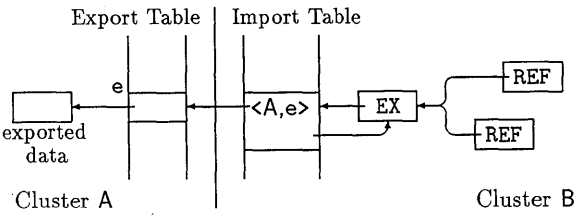


Figure 4: Export and Import Tables

During local garbage collection, data referred to by an export table entry should be regarded as active data, because it is difficult to know whether or not the export table entry is referred to by other clusters immediately. Therefore, without an efficient garbage collection scheme for the export table, many copies of non-active data would survive, these reducing the effective heap space and decreasing garbage collection performance.

One way of managing table entries efficiently is for table entries to be reclaimed incrementally. Below, we describe a method for reclaming table entries in detail.

Let us consider utilizing local garbage collection. Execution of local garbage collection might release the external reference cells. This leads to the release of import table entries and the issue of `release` messages to the corresponding export table entries. When the export table entry is no longer accessed, the entry is released. However, the reference count scheme cannot be used to manage the export table entries. This is because the increase and decrease messages for the reference counters of the export table entries are transferred through a network. Then, the arrival order of the two messages issued by the two distinct clusters is not determined in the PIM global network. This destroys the consistency of reference counters. Additionally, in the PIM network, messages are possibly overtaken. Although the reference count scheme has been improved and now requires the acknowledgment of each increase and decrease message, this will increase the network traffic.

A more efficient scheme, the *weighted export counting* (WEC) scheme has been invented [Ichiyoshi *et al.* 1988]. This is an extension of the weighted reference counting scheme [Watson and Watson 1987] [Bevan 1989] in the sense that the messages being transmitted in the loosely-coupled network also have weights. With the WEC scheme, every export table entry E holds the following invariant relation (Figure 5):

$$Weight\ of\ E = \sum_{x\ \in\ references\ to\ E} Weight\ of\ x$$

A weight is an integer. When a new export table entry is allocated, the same weight is assigned to both the export table entry and the external reference. When an import table entry is released, its weight is returned to the corresponding export table entry by the `release` message. The weight of the export table entry is decreased by the returned weight. The export table entry is detected as

no longer being accessed when the weight of the entry becomes zero. Then, the entry is released from the export table. See [Ichiyoshi *et al.* 1988] for more details on the operation of the WEC scheme.

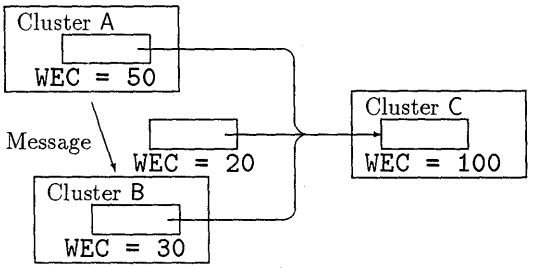


Figure 5: WEC Invariant Relation

It is important that the WEC scheme is not affected by the order in which messages arrive, and there is no need to give acknowledgment. Furthermore, the WEC scheme alleviates the cost of splitting external references.

4.2.3 Supply of Weighted Export Count

In terms of the WEC scheme, the problem of how to manage WEC when the weight of the import table entry cannot be split (when the weight reaches 1) remains.

In order to overcome this problem, we developed a *WEC supply mechanism* which is an application of the bind hook technique. The bind hook technique suspends and resumes the KL1 language (Section 2.2) [Goto *et al.* 1988].

The WEC supply mechanism works as shown in Figure 6 and 7. The current situation is that the weight of an import table entry in Cluster B reaches 1, and a goal in Cluster B issues an access command to the data in Cluster A. In this case, the message related to the access command cannot be sent, because the weight to be put on the message command cannot be got from the import table entry.

In the WEC supply mechanism, the left WEC (the weight is 1), first, is taken from the import table entry, and the import table entry is reclaimed. After that, in Cluster B, an export table entry for the external reference cell is allocated. This new external reference ID is supposed to be the return address for the reply to the following WEC supply request. At that time, the goal is hooked to the external reference cell. Eventually, Cluster B sends the `RequestWEC` message to request a new weight to Cluster A. Of course, the weight taken from the import table entry described above is returned to the corresponding export table entry by this message. Figure 6 shows the situation at that time.

When Cluster A receives the `RequestWEC` message, Cluster A adds a weight, say W, to the corresponding export table entry and returns the `SupplyWEC` message to Cluster B. The `SupplyWEC` message tells Cluster B to add the weight W to a new import table entry. In Cluster B, the suspended goal is resumed when the new import table entry is allocated. Then, the export table entry for the return address is reclaimed. Figure 7 shows the situation at that time.

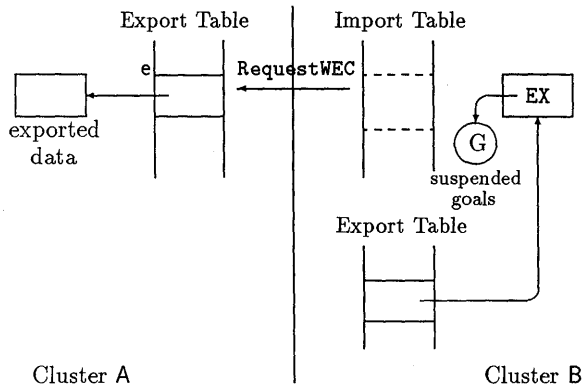


Figure 6: WEC Request Phase

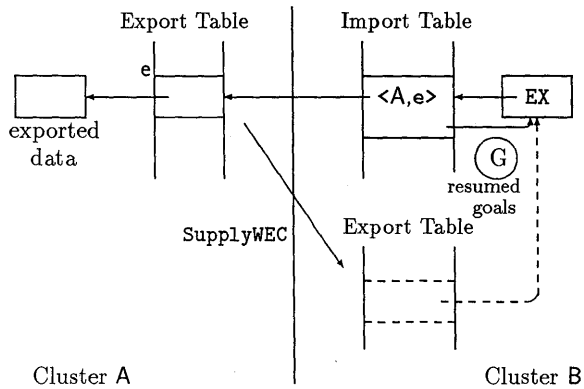


Figure 7: WEC Supply Phase

This mechanism allows the originated goal to be hooked and resumed inexpensively without additional data structures.

The KL1 language processor on Multi-PSI copes with this situation using *indirect exportation* and *zero WEC message* [Ichiyoshi *et al.* 1988]. However, the zero WEC message is a technique which is applicable to a FIFO network. As described earlier, the PIM network does not obey the FIFO rule, so the zero WEC message cannot be used in PIM. Therefore, PIM uses indirect exportation and WEC supply mechanism.

4.2.4 Mutual Exclusion of Table Entries

In order to check whether or not an external reference is already registered on the export table, a hash table is used. When an export table entry is allocated, it is registered in the hash table. When a cluster receives

a `release` message, a PE in the cluster decreases the weight of the corresponding export table entry. If the weight reaches zero, the export table entry is removed from the hash table. Figure 8 shows the data structure of the export table and its hash table. Its hash key is the address of exported datum.

Since up to about ten PEs within a cluster share these structures and access them in parallel, efficient mutual exclusion should be realized.

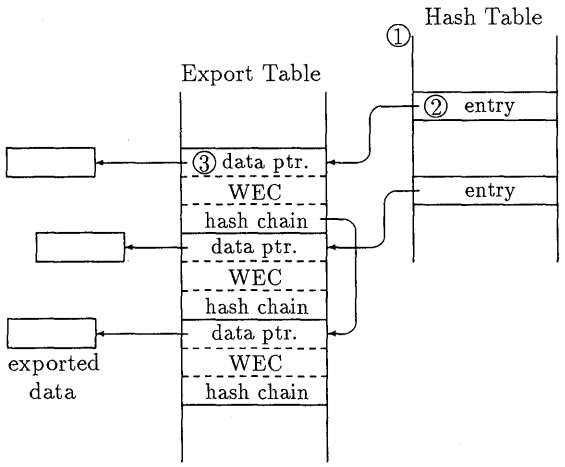


Figure 8: Data Structures of Export Table

Here, let us consider how to realize efficient mutual exclusion in the following two cases, which are typical cases of `release` message processing.

Case 1: A PE decreases the weight of an export table entry and the weight does not reach zero. In this case, only an export table entry is directly accessed. The export table entry should be locked, when manipulating its weight. The corresponding hash table entry does not need to be locked, because the hash chain does not change.

Case 2: A PE decreases the weight of export table entry and the weight reaches zero. In this case, the export table entry is released from hash table entry. Therefore, the export table entry should be locked for the same reason as in Case 1. The hash table entry should also be locked, when the export table entry is released from the hash chain, because other PEs may access the same hash chain simultaneously.

The problem is how to lock these structures efficiently. Here, we implemented the following three methods and evaluated their efficiency.

Method 1: Locking entire hash table and export table

Whenever a PE accesses the export table, the export table and the hash table are entirely locked. In Figure 8, location ① is locked.

Since the implementation of this method is simple, the total execution time is short. However, this method occupies a large locking region for a long time. Thus, access contention occurs very frequently.

Method 2: Locking one hash table entry

When a PE decreases the weight of an export table entry, the corresponding hash table entry (② in Figure 8) is locked.

In this method, the data structure to be locked is obviously smaller than in Method 1. However, this method has an overhead for computing the hash value of exported data even when the hash chain is not modified.

Method 3: Locking one hash table entry and one export table entry

When a PE decreases the weight of an export table entry, the export table entry (③ in Figure 8) is locked. If the weight becomes zero, the corresponding hash table entry (② in Figure 8) is locked. Then, the export table entry is released from the hash chain.

In this method, the locking of data structures is at a minimum and the frequency of access contention is low. However, implementation of this method is complicated.

In the above two cases, the static execution steps of the three methods are measured, using a parallel KL1 emulator on a Sequent Symmetry. Tables 2 and 3 show the results. In the tables, Total represents the total execution steps spent on receiving a `release` message. Locking region represents locking intervals, that is, how long each structure is locked.

Table 2: Locking Intervals(static steps) Case 1

	Total	Locking region		
		①	②	③
Method 1	30	23	—	—
Method 2	37	—	23	—
Method 3	32	—	0	26

Table 3: Locking Intervals(static steps) Case 2

	Total	Locking region		
		①	②	③
Method 1	61	54	—	—
Method 2	61	—	47	—
Method 3	73	—	32	27

Before evaluation, we thought that Method 1 took fewer steps than the other methods. However, there is

,actually, no great difference in the total number of execution steps. This is because the essential part of accessing the export table is complicated, and dominates the steps. In Method 1, as the ratio of the locking region to the total is relatively high, access contention to the hash table is supposed by frequent. Hence, we do not adopt Method 1.

[Takagi and Nakase 1991] tells us that WEC is effectively divided in actual programs. From this result, we assume that there are many `release` messages which just decrease the weight of WEC. That is, Case 1 occurs much more frequently than Case 2. Thus, we mostly deal with Case 1. The total execution steps of Methods 2 and 3 (37 steps and 32 steps) are almost the same, The locking intervals of Methods 2 and 3 (23 steps and 26 steps) are almost the same. It is preferable that the data structure to be locked is small. According to this discussion, we adopt Method 3 as the mutual exclusion method for the export table.

For the import table, a similar technique is used to reclaim the import table entries.

4.3 Parallel Copying Garbage Collector

Efficient garbage collection (GC) methods are especially crucial for the KL1 language processor on multiprocessor systems. Since the KL1 execution dynamically consumes data structures, GC is necessary for reclaiming storage during computation. Moreover, GC should be executed at each cluster independently since it is very expensive to synchronize all clusters.

As we described briefly in Section 3, an incremental GC method based on the MRB scheme was already proposed and implemented on Multi-PSI [Inamura *et al.* 1988], however since it cannot reclaim all garbage objects, it is still important to implement an efficient GC to supplement MRB GC.

We invented a new parallel execution scheme of stop and copy garbage collector, based on Baker's sequential stop-and-copy algorithm[Baker 1978] for shared memory multiprocessors. The algorithm allocates two heaps although only one heap is actively used during program execution. When one heap is exhausted, all of its active data objects are copied to the other heap during GC. Thus, since Baker's algorithm accesses active objects this algorithm is simple and efficient.

Innovative ideas in our algorithm are the methods which reduce access contention and distribute work among PEs during cooperative GC. Also no inter-cluster synchronization is needed since we use the export table described in Section 4.2. A more detailed algorithm is described in [Imai and Tick 1991].

4.3.1 Parallel Algorithm

Parallelization: There is potential parallelism inherent in the copying and scanning actions of Baker's algorithm, i.e., accessing S and B. Here pointer S represents the *scanning point* and B points to the *bottom* of the new heap. A naive method of exploiting this parallelism is to allow multiple PEs to scan successive cells at S, and copy them into B. Such a scheme is bottlenecked by the PEs vying to atomically read and increment S by one cell and atomically write B by many cells. Such a contention is unacceptable.

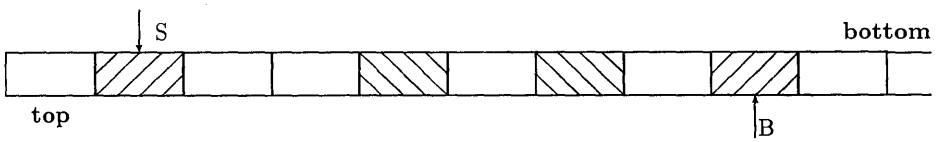
Private Heap: One way to alleviate this bottleneck is to create multiple heaps corresponding to multiple PEs. This is the structure used in both Concert Multilisp[Halstead 1985] and JAM Parlog[Crammond 1988] garbage collectors. Consider a model where each PE(i) is allocated private sections of the new heap, managed with private S_i and B_i pointers. Copying from the old space could proceed in parallel with each PE copying into its private new sections. As long as the *mark* operation in the old space is atomic, there will be no erroneous duplication of cells. Managing private heaps during copying, however, presents some significant design problems:

- Allocating multiple heaps within the fixed space causes fragmentation.
- It is difficult to distribute the work among the PEs throughout the GC.

To efficiently allocate the heaps, each PE extends its heap incrementally in *chunks*. A chunk is defined as a unit of contiguous space, that is a constant number of HEU cells (HEU ≡ Heap Extension Unit). We first consider a simple model, wherein each PE operates on a single heap, managed by a single pair of S and B pointers. The B_{global} pointer is a state variable pointing to the global bottom of the new allocated space shared by all PEs. Allocation of new chunks is always performed at B_{global}.

Global Pool for Discontiguous Areas: When a chunk has been filled, the B pointer reaches the top of the next chunk (possibly not its own!). At this point a new chunk must be allocated to allow copying to continue. There are two cases where B overflows: either it overflows from the same chunk as S, or it overflows from a discontiguous chunk. In both cases, a new chunk is allocated. In the former case, nothing more needs to be done because S points into B's previous chunk, permitting its full scan. However, in the latter case, B's previous chunk will be lost if it is separated from S's by extraneous chunks (of other PEs, for instance).

The problem of how to 'link' the discontiguous areas, to allow S to freely scan the heap, is solved in the following manner. In fact, the discontiguous areas are not



The shaded portions of the heap are owned by a PE(i) which manages S and B. Other portions are owned by any PE(j) where $j \neq i$. The two chunks shaded as '/' are referenced by PE(i) via S and B. The other chunks belonging to PE(i), shaded as '\', are *not* referenced. To avoid losing these chunks , they are registered in the global pool.

Figure 9: Chunk Management in Simple Heap Model

linked at all. When a new chunk is allocated, the B's previous chunk is simply added to a *global pool.* This pool holds chunks for load distribution, to balance the garbage collection among the PEs. Unscanned chunks in the pool are scanned by idle PEs which resume work (see Figure 9).

Uniform Objects in Size: We now extend the previous simple model into a more sophisticated scheme that reduces the fragmentation caused by dividing the heap into chunks of uniform size. Imprudent packing of objects which come in various sizes into chunks might cause fragmentation, leaving useless area in the bottom of chunks. To avoid this problem, each object is allocated the closest quantum of 2^n cells (for integer $n <$ log(HEU)) that will contain it. Larger objects are allocated the smallest multiple of HEU chunks that can contain them. When copying objects, smaller than HEU, into the new heap, the following rule is observed: "All objects in a chunk are always uniform in size." If HEU is an integral power of two, then no portion of any chunk is wasted. When allocating heap space for objects of size greater than one HEU, contiguous chunks are used.

In this refined model, chunks are categorized by the size of the objects they contain. To effectively manage this added complexity, a PE manipulates multiple $\{S, B\}$ pairs (called $\{S_1, B_1\}$, $\{S_2, B_2\}$, $\{S_4, B_4\}$, ..., and $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$). Initially, each PE allocates multiple chunks with S_i and B_i set to the top of each chunk.

Referring back to Figure 9, recall that shaded chunks of the heap are owned by PE(i) and non-shaded chunks are owned by other PEs. The chunks shaded as '/', in the extended model, contain objects of some fixed size k, and are managed with a pointer pair $\{S_k, B_k\}$. Chunks shaded as '\' are either directly referenced by other pointer pairs of PE(i) (if they hold objects of size $m \neq k$), or are kept in the global pool.

Load Balancing: In the previous algorithm, it is a difficult choice to select an optimal HEU. As HEU increases, B_{global} accesses become less frequent (which is desirable, since contention is reduced); however, the average distance between S and B (in units of chunks) decreases. This means that the chance of load balancing decreases with increasing HEU.

One solution to this dilemma is to introduce an independent, constant size unit for load balancing. The load distribution unit (LDU) is this predefined constant, which is distinct from HEU[10] and enables more frequent load balancing during GC. In general, the optimized algorithm incorporates a new rule, wherein if $(B_k - S_k > LDU)$, then the region between the two pointers (i.e., the region to be scanned later) is pushed onto the global pool.

4.3.2 Evaluation

The parallel GC algorithm was evaluated for a large set of benchmark programs (from [Tick 1991] etc.) executing on a parallel KL1 emulator on a Sequent Symmetry. Statistics in the tables where measured on eight PEs with HEU=256 words and LDU=32 words, unless specified otherwise. A more detailed evaluation is given in [Imai and Tick 1991].

To evaluate load balancing during GC, we define the *workload* of a PE and the *speedup* of a system as follows:

$$\begin{aligned} \text{workload(PE)} &= \text{number of cells copied} + \\ &\quad \text{number of cells scanned} \\ \text{speedup} &= \frac{\sum \text{workloads}}{max(\text{workload of PEs})} \end{aligned}$$

The workload value approximates the GC time, which cannot be accurately measured because it is affected by DYNIX scheduling on Symmetry. Workload is measured in units of *cells referenced.* Speedup is calculated with the assumption that the PE with the *maximum* workload determines the *total* GC time. Note that speedup only represents how well load balancing is performed and does not take into account any extra overheads of load balancing (which are tackled separately). We also define the *ideal speedup* of a system:

$$\text{ideal speedup} = min\left(\frac{\sum \text{workloads}}{max(\text{workload for one object})}\ ,\ \#\text{PEs}\right)$$

[10] We assume that HEU = kLDU, for integer $k > 0$.

Benchmark	avg. WL ×1000	Speedup Size of LDU 32w	64w	128w	256w	ideal
BestPath	165	7.15	7.06	6.46	6.36	8.00
Boyer	47	5.67	5.83	4.38	4.12	8.00
Cube	139	7.74	7.67	7.35	6.83	8.00
Life	101	7.10	6.86	6.31	6.29	8.00
MasterMind	4	2.50	2.48	2.58	2.48	2.87
MaxFlow	95	4.06	3.84	3.70	2.86	8.00
Pascal	5	2.67	2.91	3.45	2.77	7.25
Pentomino	3	4.34	3.34	3.67	4.21	8.00
Puzzle	17	2.63	2.84	2.58	2.61	2.92
SemiGroup	496	7.75	7.28	7.49	7.02	8.00
TP	17	2.49	2.39	2.43	2.33	2.79
Turtles	203	7.79	7.44	7.20	7.22	8.00
Waltz	32	4.38	2.92	2.31	1.64	8.00
Zebra	167	6.27	6.04	6.42	6.28	8.00

Table 4: Average Workload and Speedup (8 PEs, HEU=256 words)

Benchmark	LDU (words) 32	64	128	256
BestPath	421.0	139.6	84.4	45.8
Boyer	208.8	131.3	24.3	12.8
Cube	609.4	241.6	96.3	55.5
Life	145.8	66.5	29.8	14.8
MasterMind	3.9	1.5	1.1	1.0
MaxFlow	211.3	75.0	37.0	10.0
Pascal	1.6	1.0	1.0	1.0
Pentomino	134.3	65.3	21.0	7.5
Puzzle	51.6	30.6	10.5	4.9
SemiGroup	1,700.7	910.8	439.3	29.6
TP	44.4	19.8	8.8	4.6
Turtles	1,427.0	640.0	314.0	136.0
Waltz	76.0	36.0	11.5	1.4
Zebra	2,127.9	920.2	467.7	222.4

Table 5: Accesses of the Global Pool (8 PEs, HEU=256 words)

Ideal speedup is meant to be an approximate measure of the fastest that n PEs can perform GC. Given a perfect load distribution where $1/n$ of the sum of the workloads is performed on each PE, the ideal speedup is n. There is an obvious case when an ideal speedup of n cannot be achieved: when a single data object is so large that its workload is greater than $1/n$ of the sum of the workloads. In this case, GC can complete only after the workload for this object has completed. These intuitions are formulated in the above definition.

Speedup: Table 4 summarizes the average workload and speedup metrics for the benchmarks. The table shows that benchmarks with larger workloads display higher speedups. This illustrates that the algorithm is quite practical. It also shows that the smaller the LDU, the higher the speedup obtained. This means there are the more chances to distribute unscanned regions, as we hypothesized.

In some benchmarks, such as MasterMind, Puzzle and TP, ideal speedup is limited (2–3). This limitation is due to an inability of PEs to cooperate in accessing a single large structure. The biggest structure in each of the benchmark programs is the *program module*. A program module is actually a first-class structure and therefore subject to garbage collection (necessary for a self-contained KL1 system which includes a debugger and incremental compiler). In practice, application programs consist of many modules, opposed to the benchmarks measured here, with only a single module per program. Thus the limitation of ideal speedup in MasterMind and Puzzle is peculiar to these toy programs.

In benchmarks such as Pascal and Waltz, the achieved speedup is significantly less than the ideal speedup. These programs create many long, flat lists. When copying such lists, S and B are incremented at the same rate. The proposed load distribution mechanism does not work well in these degenerate cases. Our method works best for deeper structures, so that B is incremented at a faster rate than S. In this case, ample work is uncovered and added to the global pool for distribution.

Contention at the Global Heap Bottom: We analyzed the frequency with which the global heap-bottom pointer, B_{global}, is updated (for allocation of new chunks). This action is important because B_{global} is shared by all the PEs, which must lock each other out of the critical sections that manage the pointer. For instance, in Zebra (given HEU = 256 words and LDU = 32 words), B_{global} is updated 3,885 times by GCs. If B_{global} were updated whenever a single object was copied to the new heap, the value would be updated 126,761 times. Thus, the update frequency is reduced by over 32 times compared to this naive update scheme. In other benchmarks, the ratios of the other programs range from 15 to 114.

Global-Pool Access Behavior: Table 5 shows the average number of global-pool accesses made by the benchmarks, and the average number of cells referenced (in thousands) by the benchmarks per global-pool access. These statistics are shown with varying LDU sizes. The data confirms that, except for Pascal and MasterMind, the smaller the LDU, the more chances these are to distribute unscanned regions, as we hypothesized. The amount of distribution overhead is at least two orders of magnitude below the useful GC work, and in most cases, at least three orders of magnitude below.

As described above, to achieve efficient garbage collection on a shared-memory multiprocessor system, load distribution and the working set size should also be carefully considered.

4.4 Goal Scheduling in a Cluster

An efficient goal scheduling algorithm within a cluster must satisfy the following criteria:

1. no idle processing elements
2. high data locality
3. less access contention
4. no disturbance of busy processing elements

Moreover, since the KL1 language has the concept of goal priority (Section 3.1.3), goals with higher priorities within a cluster are the targets of scheduling. Notice that *Load* is the amount of work to be performed by a PE, cluster or system. Thus, load does not mean the number of goals.

No Idle Processing Elements: The aim of goal scheduling is to finish the execution of application programs earlier. Previous software simulation told us the following [Sato and Goto 1988]:

- To keep all PEs busy is the most effective way of load balancing since the goals of the KL1 language are, in general, fine-grained and have rich parallelism.
- Making the numbers of goals of each PE the same during execution does not lead to good load balancing.

Here, an idle PE means one that does not have any goals to be reduced, or one that reduces goals with lower priorities.

High Data Locality: Since a cluster is viewed as a shared-memory multiprocessor, it is important to keep the data locality high to achieve high performance. This means keeping the hit ratio of snooping caches high. In our KL1 runtime system, once argument data are allocated to a memory, the locations are not moved (only a garbage collector can move them). Hence, it is desirable that a goal that includes references to the argument data is reduced by a PE in which the cache already contains the data. Furthermore, in terms of KL1 goal reduction, suspension and resumption during unification give rise to expensive context switching. If context switching occurs frequently, the hit ratio of snooping caches decreases and, consequently, the total performance is seriously degraded.

Less Access Contention: To schedule goals properly, each PE has to access shared resources in parallel. For instance, there is a goal pool that stores goals to be reduced and priority information that must be exchanged among PEs. Since expensive mutual exclusion is required when PEs within a cluster access these shared resources, access conflicts should be decreased as much as possible.

No Disturbance of Busy Processing Elements: From the load balancing point of view, it is better to have as many idle PEs as possible involved in work associated with goal scheduling. Moreover, when an idle PE tries to find a new goal, it is desirable that the idle PE should neither interrupt nor disturb the execution of busy PEs.

Consequently, well-distributed data structures and algorithms should be designed so that these criteria are satisfied as much as possible.

4.4.1 Goal Pool

Let us consider two ways of implementing a goal pool: centralized implementation and distributed implementation. That is, one queue in a cluster or one queue for every PE. If centralized implementation is used, priority is strictly managed. However, every time a goal is picked up and new goals are stored, the access contention may occur. Thus, our KL1 implementation adopts the distributed implementation method. It turns out that transmission of goals between PEs for load balancing is required and priority is loosely managed. On the contrary, however, distributed queue management is necessarily loose for priority.

The distributed goal queues are managed using a depth-first rule to keep the data locality high. Under depth-first (LIFO) management, it is presumed that the same PE will often write and read the same data and that the number of suspensions and resumptions invoked will be less. Therefore, the cache hit ratio increases.

Further, when a PE resumes goal unification, the PE sends the goal to the queue of the PE which suspended the goal previously. This also contributes to keeping the data locality high.

As described above, since goals are accompanied with priorities, in our KL1 implementation, a PE has its own goal queues for each priority. Figure 10 shows the goal queues with priorities.

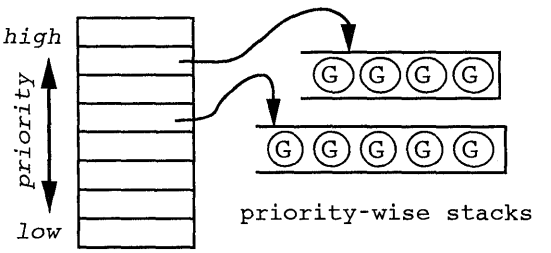


Figure 10: Goal Queue with Priorities

4.4.2 Transmission of Goals

As soon as a PE becomes or may become idle, it must take a new goal with higher priority from the queue of a PE with a small overhead to avoid going into an idle state. An idle PE triggers the transmission of a new goal.

Here, two design decisions are needed. One decision is deciding whether the PE that transmits a new goal with high priority is a request sender (idle PE) or a request receiver (busy PE). Another decision is deciding whether a new goal is to be picked from the top of a queue or the end. If an idle PE has the initiative, access contention may occur in the queue of a busy PE. If a busy PE has the initiative, the CPU time of the busy PE must be consumed. If a new goal is picked from the top of a queue, it may destroy the data locality of the busy PE's cache. If a new goal is at the end, it will often happen that the goal reduction of an idle PE is immediately suspended; the potential load of the goal may be small under LIFO management. Thus, this method may frequently trigger transmission.

The current implementation uses dedicated PIM hardware which broadcasts requests to all PEs within a cluster, in order to issue a request for a new goal to the other PEs. Each busy PE executes an event handler once a reduction and the event handler may catch the request. Then, the busy PE which catches the request first picks up the goal with the highest priority from the top of its goal queue. Our implementation should be evaluated for comparison.

4.4.3 Priority Balancing

A PE always reduces goals which belong to its local queue and have the highest priority. There are two problems; one is how to detect the priority imbalance, and the other is how to correct the imbalance by cooperating with the other PEs. Our priority balancing scheme was designed so that fewer shared resources are required and busy PEs do less work concerned with priority balancing (Figure 11). Our scheme requires only one shared

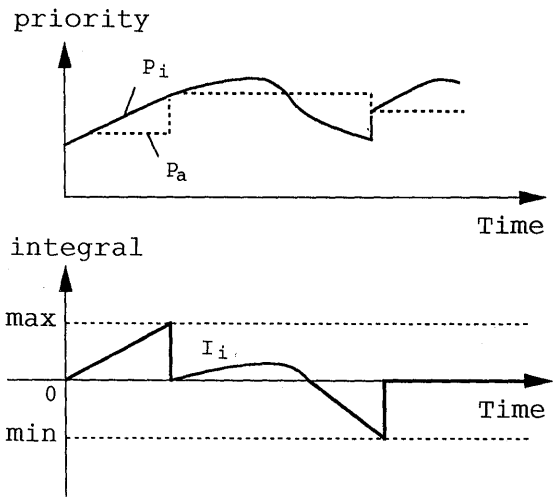


Figure 11: Priority Balancing Scheme

variable P_a to record an average priority, and the same number of variables $I_1 \sim I_n$ as the number of PEs to record a current integral value for each PE. A current priority of each PE is represented by P_i. There are two constants, max (> 0) and min (< 0). Every PE will always calculate the integral I_i of $P_i - P_a$ along time. When $I_i > max$, the PE(i) adjusts P_a to the current P_i and resets I_i to zero. When $I_i < min$, the PE(i) issues a goal request, adjusts P_a to the priority of a transmitted goal, and resets I_i to zero. The mechanism of the goal transmission described above is used as well, since the goal with the highest PE priority is picked up. More details on this algorithm are described in [Nakagawa *et al.* 1989].

The features of this scheme are as follows. The calculation of the integral reduces the frequency of shared resource P_a updating and busy PEs do some work only when $I > max$.

The disadvantages are as follows. It may happen that the priority of a transmitted goal is even lower, that P_a decreases unreasonably, and that the frequency of the high-priority goal transmission decreases. Our priority balancing scheme utilizes the goal transmission mechanism (Section 4.4.2), which does not always transfer the goal with the most appropriate priority. Accordingly, a load imbalance may be sustained for a while. How well this method works depends on the priority of the goals transmitted upon requests. In other words, there is a tradeoff between loose priority management and the frequency of high-priority goal transmission. Further, in this scheme, a busy PE (a PE satisfying $I_i > max$) has to write its current priority P_i to the shared variable P_a. This may cause access conflict and disturb the busy PE.

A new scheme which we will design should overcome these problems. However, we think that calculation of the integral along time is essential even in new schemes.

4.5 Meta Control Facilities

When designing the implementation for a shoen, we assume that the following dynamic behavior applies in the KL1 system:

- Shoen statuses change infrequently.
- Shoen operations are not executed immediately but within a finite time.
- Messages transferred are possibly overtaken in the inter-cluster network.

Under these assumptions, our implementation must satisfy the following requirements:

- The less inter-cluster messages the better.
- No bottleneck appears; algorithms and protocols that do not frequently access shoen records and foster-parent records are desirable.
- The processing associated with meta control should not degrade the performance of reduction.

Many techniques realizing a shoen have been developed to achieve high efficiency. This section concentrates on execution control and resource management.

From now on, stream messages on the control and report streams for communication to the outside are represented in a typewriter typeface, such as `start`, `add_resource`, and `ask_statistics`.

4.5.1 Execution Control

This section describes schemes for implementing the functions for execution control. Schemes (1) $\sim$ (2) are effective in a shared-memory environment (intra-cluster). Schemes (3) $\sim$ (5) are effective in a distributed-memory environment (inter-cluster).

(1) Change of Foster-parent Status: Since goal reduction cannot be started when the status of foster-parent which the goal belongs to is not *started*, imprudent implementation needs to check the status of a foster-parent before every goal reduction. To avoid such frequent checking, a status change of the foster-parent is notified by the interruption mechanism. When a cluster receives a message that changes a foster-parent's status to non-executable, an interruption is issued to every PE in the cluster. When a PE catches the interruption, the PE checks to see if the current goal belongs to the target foster-parent. If so, then the foster-parent is to be stopped and the PE suspends execution of the current goal and starts to reduce the goal of the other active foster-parent. Otherwise, the PE continues the reduction. Since the newly scheduled goal is supposed to belong to the other foster-parent, the context of the goal reduction [11] must be switched, too.

The assumption that the status of a foster-parent is switched infrequently implies that interruptions happen rarely. Thus, an advantage of the scheme is that the ordinary reduction process rarely suffers from foster-parent checking.

(2) Foster-parent Termination Detection: To detect the termination of a foster-parent efficiently, a counter called *childcount* is introduced. The childcount represents the sum of both the number of goals and the number of shoens which belong to the foster-parent. When the childcount of a foster-parent reaches zero, all goals of the foster-parent are finished.

The childcount area is allocated in a foster-parent record, and all PEs in a cluster must access the area. Since this counter must be updated whenever a goal is created or terminated, frequent exclusive updating of this counter might become a bottleneck. To reduce such an access contention, the cache area of the childcount is allocated on each PE. The operations go as follows. At first, a counter is allocated on the childcount cache of each PE, initialized with a value zero. Every time a new goal is spawn, the counter is incremented, and the counter is decremented upon the end of goal reduction. When the reduction of a new goal whose foster-parent differs from the previous one begins, the current foster-parent should be switched. That is, the value of the counter on the childcount cache is brought back to the previous foster-parent record, and the counter is reinitialized. The foster-parent terminates when it detects that the counter on the foster-parent record is zero.

This scheme is expected to work efficiently if foster-parents are not changed often.

(3) Point-to-point Message Protocol: Basically, message protocols based on point-to-point communication between a shoen and a foster-parent are not designed on the basis of broadcasting [Rokusawa *et al.* 1988]. If almost all clusters always contain foster-parents of a shoen, protocols based on broadcast are taken into account. However, the current implemetation does not assume this, although it depends on applications. Therefore, it is inefficient to broadcast messages to all clusters in the system every time. Then, a shoen provides a table that indicates whether or not its foster-parent exists in a cluster corresponding to the table position. The table is maintained by receiving foster-parent creation and termination messages from the other clusters. Accordingly, a shoen can send messages only to the clusters where its foster-parents reside.

(4) Lazy Management of Foster-parent: A shoen controls its foster-parents by exchanging messages, such as start/stop messages. However, these messages may overtake, and, thus, a foster-parent may go into the incorrect states. For the stats to be correct and to minimize the maintenance cost, received start/stop messages are managed by a counter. If a start message arrives, the foster-parent increments the counter. If a stop message arrives, the foster-parent decrements the counter. Then, when the counter value crosses zero, the foster-parent changes the execution status properly.

(5) Shoen Termination Detection: To detect the termination of a shoen efficiently, a Weighted Throw Count (WTC) scheme was introduced [Rokusawa *et al.* 1988] [Rokusawa and Ichiyoshi 1992]. This scheme is also an application of the weighted reference count scheme [Watson and Watson 1987][Bevan 1989]. Logically, a shoen is terminated when there are no foster-parents. However, this is not correct enough to maintain the number of foster-parents, since goals thrown by a foster-parent may be transferred in the network. Thus, a foster-parent lets both all goals to be thrown and all messages between a shoen and foster-parents to have a portion of the foster-parent's weight. On terminating a foster-parent, all foster-parent weights are returned to

[11] A childcount cache and a resource cache.

the shoen. If the foster-parent terminated at message arrival, the messages from the shoen are also sent back to the shoen to keep its weight. Then, when all weights are returned to the shoen, the shoen terminates itself. An advantage of this scheme is that it is free from sending acknowledgement messages.

Thus, since a shoen must not continue to lock shared resources in this scheme until an acknowledgement returns, the scheme can reduce not only the network traffic but can also alleviate mutual exclusion.

4.5.2 Resource Management

As described above, a shoen is also used as a unit for resource management. In the KL1 language, the reduction time is regarded as the computation resource. The shoen consumes the supplied resources while shifting the resources. Moreover, since a shoen works in parallel, lazy resource management is inevitable, like in the shoen execution control (Section 4.5.1).

A shoen has a limited amount of resources which it can consume. Upon exceeding the limit, goals in the shoen cannot be reduced. When a runtime system detects that the total amount of consumed resources so far is approaching the limit, a `resource_low` message is automatically issued on the shoen's report stream. The shoen stops its execution with its resources exhausted. On the other hand, the `add_resource` message on the control stream raises the limit and the shoen can utilize the resource up to the new limit. Furthermore, a shoen which accepts the `ask_statistics` message reports the current resources consumed so far.

This section describes our resource management implementation schemes.

(1) Distributed Management: The scheme is briefly described below. Figure 12 shows the resource flow between a shoen and its foster-parents.

A shoen has a limit value, which indicates that the shoen can consume resources up to the limit. Initially, the resource limit is zero. Only the `add_resource` message can raise the limit. When a shoen receives the `add_resource` message, the shoen requests new resources to the above foster-parent by a value within the limit value designated by the `add_resource` message. Here, we also call this foster-parent the *parent* foster-parent. Notice that a shoen and its parent foster-parent reside in the same cluster, and, thus, the operation for the resource request is implemented by read and write operations on a shared memory.

After a shoen has got new resources from its parent foster-parent, the shoen further supplies resources to its foster-parents which requested resources by the supply_resource message across clusters. Moreover the supplied resources may be supplied to the descendant shoens and foster-parents. Then, those foster-parents

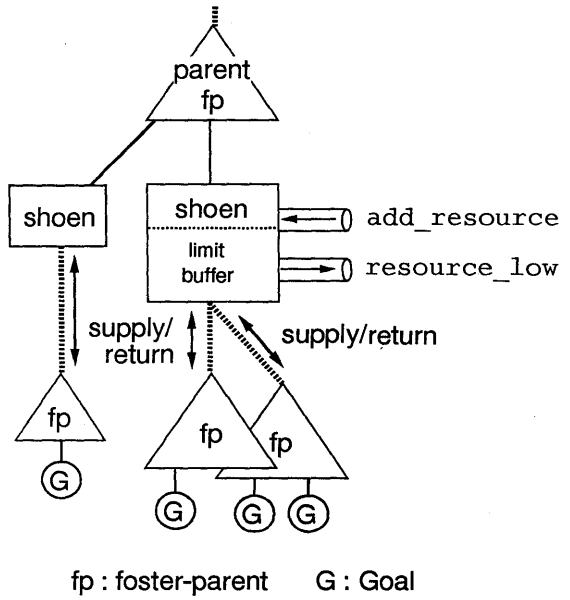


Figure 12: Resource Flow Between a Shoen and its Foster-parents

consume the supplied resources. The shoen has a buffer for the resources; the excessive resources returned from terminated foster-parents are stored in the shoen buffer. When the remaining resources of a foster-parent are going to run out, a resource request message is sent to the above shoen. If the shoen cannot afford to supply the requested resources, the shoen issues the `resource_low` message on its report stream. Otherwise, if the shoen can afford and has sufficient resources in the buffer, the resources are supplied to the foster-parent immediately. If there are insufficient resources, the shoen requests new resources within the current limit value from its parent foster-parent. As described here, the resource buffer of a shoen can prevent the message from being issued more frequently than necessary.

If the resources of the foster-parent are exhausted, goal reduction stops. Then, the scheduled goals are hooked on to the foster-parent record, in preparation for rescheduling when new resources are supplied from the shoen.

Furthermore, each PE has a resource cache area for the foster-parent, and, hence, a counter is actually decremented every time a goal is reduced. This mechanism is similar to the childcount mechanism (Section 4.5.1). However, when the foster-parent of a goal to be reduced alters, the caches on PEs must be brought back to the foster-parent record.

(2) Resource Statistics: While the system enjoys lazy resource management, it gets harder to collect resource information over the entire system. A shoen receives the `ask_statistics` message, which reports the current total consumed resources.

The scheme used to collect the information is described. A shoen issues inquiry messages to each foster-parent. When an inquiry message arrives at a foster-parent, the foster-parent informs each PE of this using the interruption mechanism. This portion is similar to the mechanism of Section 4.5.1 (1). The PEs which catch the interruption check if the current goals belong to the target foster-parent. If so, the PE puts the resource on the cache back to the foster-parent record. When all corresponding PEs have been put back, the subtotal resource on the foster-parent appears. If not, the PEs do nothing and reduction continues. Then, the foster-parent reports the subtotal to the shoen and re-distributes some resources back to the PEs. As a result, the PEs resume goal reduction.

We assume that the `ask_statistics` message is issued infrequently. This scheme works well.

(3) Point-to-point Resource Delivery: The destination of new resources when a shoen receives resource request messages from its foster-parents is a design decision. It must be decided whether the shoen delivers the new resources only to the foster-parents which have requested them, or delivers them to all foster-parents. A protocol based on broadcast may be preferable when the foster-parents in nearly all clusters always possess the same amount of resources and consume them at the same speed. The current method is similar to one in Section 4.5.1 (3).

Our assumptions we based on an experience of the Muti-PSI system. Goal scheduling within a cluster, however, differs and there is no guarantee that every cluster has the foster-parent of the shoen. Therefore, in the current implementation method the shoen sends the resource supply message just to the clusters which have sent resource request messages.

4.6 Intermediate Instruction Set

The KL1 compiler for PIM has two phases. The first phase compiles a KL1 program into an intermediate instruction code; the instruction set is called KL1-B. The second phase translates the intermediate code into a native code. KL1-B is designed for an *abstract KL1 machine* [Kimura and Chikayama 1987], interfacing between the KL1 language and PIM hardware, just as in Warren Abstract Machine [Warren 1983] of Prolog.

KL1-B for PIM is extended from KL1-B for Multi-PSI to efficiently exploit the PIM hardware.

4.6.1 Abstract KL1 machine

The *abstract KL1 machine* is simple virtual hardware to describe a KL1 execution mechanism. It has a single PE with a heap memory and basically expresses the inside execution of a PE. However, every KL1-B instruction implicitly supports multi-PE processing. Further, some KL1-B instructions are added for inter-*cluster* processing.

A goal is represented by a *goal record* on a heap. The *goal record* consists of arguments and an execution environment which includes the number of arguments and the address of the predicate code. A ready goal is managed in the *ready goal pool* which has entries for each priority. Each entry indicates a linked stack of *goal records*. Suspended goals are *hooked* on the responsible variable.

Each data word consists of a value part, a type part and a MRB part [Chikayama and Kimura 1987]. An MRB part is valid, if the value part is a pointer, and indicates whether its object is single-referenced or multiple-referenced. It is used for incremental garbage collection and destructive structure updating.

4.6.2 Overview of KL1-B

The intermediate instruction set KL1-B was designed according to the following principles:

- Memory based scheme — goal arguments are basically kept on a goal record at the beginning of a reduction, and each of them is read onto a register explicitly just before it is demanded. Thus, almost all registers are used temporarily (Section 4.6.3).
- Optimization using the MRB scheme — some instructions to reuse structures are supported to alleviate execution cost (Section 4.6.4).
- Clause indexing — the compiler collects the clauses which test the same variables, and compiles them into an instruction module. Then, all guard parts of a predicate are compiled as one into the code with branch instructions forming a tree structure (Section 4.6.5).
- Each body is compiled into a sequence of instructions which run straight ahead without branching.

The basic KL1-B instruction set is shown in Table 6.

4.6.3 Memory Based Scheme

The Multi-PSI system executes a KL1 program using the *register based scheme* — all arguments of the current goal are loaded onto *argument registers* before reduction begins, just as WAM does for Prolog.

Here, let us compare the following two methods in terms of the argument manipulation cost:

- In the *memory based scheme*, the arguments referred to in the reduction are loaded and the modified arguments are stored at every reduction. There is no cost for goal switching.
- In the *register based scheme*, all arguments of the swapped out goal are stored and all arguments of the swapped in goal are loaded at every goal switching.

Table 6: Basic KL1-B Instruction Set

KL1-B Instruction		Specification
For passive unification:		
load_wait	*Rgp,Pos,Rx,Lsus*	Read a goal argument onto *Rx* and check binding.
read_wait	*Rsp,Pos,Rx,Lsus*	Read a structure element onto *Rx* and check binding.
is_atom/integer/list/...	*Rx,Lfail*	Test data type of *Rx*.
test_atom/integer	*Rx,Const,Lfail*	Test data value of *Rx*.
equal	*Rx,Ry,Lsus,Lfail*	General unification.
suspend	*Lpred,Arity*	Suspend the current goal
For argument/element preparation:		
load	*Pgp,Pos,Rx*	Read a goal argument onto *Rx*.
read	*Rsp,Pos,Rx*	Read a structure element onto *Rx*.
put_atom/integer	*Const,Rx*	Put the atomic constant onto *Rx*.
alloc_variable	*Rx*	Allocate a new variable and put the pointer onto *Rx*.
alloc_list/vector	*(Arity,)Rx*	Allocate a new list/vector structure and put the pointer onto *Rx*.
write	*Rx,Rsp,Pos*	Write *Rx* onto a structure element.
For incremental garbage collection:		
mark	*Rx*	Mark MRB of *Rx*.
collect_value	*Rx*	Collect the structure recursively unless its MRB is marked.
collect_list/vector	*(Arity,)Rx*	Collect the list structure unless its MRB is marked.
reuse_list/vector	*(Arity,)Rx*	*collect_list/vector* + *alloc_list/vector*.
For active unification:		
unify_atom/integer	*Const,Rx*	Unify *Rx* with the atomic constant.
unify_bound_value	*Rsp,Rx*	Unify *Rx* with the newly allocated structure.
unify	*Rx, Ry*	General unification.
For goal manipulation and event handling:		
collect_goal	*Arity,Rgp*	Reclaim the *goal record*.
alloc_goal	*Arity,Rgp*	Allocate a new *goal record*.
store	*Rx,Rgp,Pos*	Write *Rx* onto a goal argument.
get_code	*CodeSpec,Rcode*	Get the code address of the predicate onto *Rcode*.
push_goal	*Rgp,Rcode,Arity*	Push the goal to the current priority entry of *ready goal pool*.
push_goal_with_priority	*Rgp,Rcode,Rprio,Arity*	Push the goal to the specified priority entry of *ready goal pool*.
throw_goal	*Rgp,Rcode,Rcls,Arity*	Throw the goal to the specified *cluster*.
execute	*Rcode,Arity*	Handle the event if it occurrs and execute the goal repeatedly.
proceed		Handle the event if it occurrs and take a new goal from *ready goal pool* to start the new reduction.

Some arguments may be moved between registers at every reduction.

Therefore, the *memory based scheme* is better than the *register based scheme* when

- Goal switching occurs frequently.
- A goal has many arguments.
- A goal does not refer to many arguments in a reduction.

Actually, these cases are expected to be seen often in large KL1 programs. Thus, we have to verify the *memory based scheme* with many practical KL1 applications.

Additionally, the number of goal arguments is limited to the number of *argument registers* — 32 in the case of Multi-PSI. This limitation is too tight and is not favorable to KL1 programmers. The *memory based scheme* can alleviate this limitation to some extent. On the other hand, the naive *memory based scheme* necessarily writes back all arguments to the *goal record*, even if tail recursion is employed. Since this is very wasteful, an optimization to keep frequently referenced arguments on registers is mandatory during tail recursion.

4.6.4 Optimization

Two optimization techniques are introduced: tail recursive optimization and the reuse of data structures. We can describe these using the following sample codes.

- source code:

```
app([H|L],T,X) :- true | X=[H|Y], app(L,T,Y).
app([ ]   ,T,X) :- true | X=T.
```

- intermediate code:

```
app_entry:
  load          CGP, 0, R1      % Load up
```

```
  load              CGP, 2, R2      % arguments
app_loop:
  wait              R1, sus_or_fail
  is_list           R1, next
  commit
* read              R1, car, R3     % H
  read              R1, cdr, R4     % L
  reuse_list        R1
* write             R3, R1, car     % H
  alloc_variable    R5              % Y
  write             R5, R1, cdr
  unify_bound_value R1, R2
  move              R4, R1
  move              R5, R2
  execute_tro       app_loop
      ⋮
next:
  is_atom           R1, sus_or_fail
  test_atom         [], R1
  commit
  load              CGP, 1, R3      % T
  unify             R3, R2
  collect_goal      3, CGP
  proceed
sus_or_fail:
  store             R1, CGP, 0      % Write back
  store             R2, CGP, 2      % arguments
  suspend           app_entry, 3
```

Tail Recursive Optimization: Some instructions are added for this optimization. *Wait* tests if an argument on a register is instantiated. *Move* prepares arguments for the next reduction. *Execute_tro* executes a goal while some arguments are kept on registers.

In the above source code, the first and third arguments of the first clause are used in tail recursion. These arguments are loaded at the beginning of the reduction by the *load* instructions which are placed before the tail recursive loop. There is no need to write them into the *goal record* during tail recursion. However, they must be written back to the *goal record* explicitly before, say, switching the goal caused by the *suspend* instruction. Since the second argument is not used in tail recursion, it is kept on the *goal record* until it is referred to in the second clause.

In this example, two *write* instructions and two *read* instructions are replaced with two *move* instructions. Thus, by assuming a cache hit ratio of 100 %, this optimization can save two steps on each recursion loop.

Reuse of Data Structures: KL1-B for PIM supports the reuse of data structures. The *reuse_list* and *reuse_vector* instructions realize this. These instructions reuse an area in a heap on which the structure unified in a guard part was allocated, but, only if the MRB of the reference to the area is not marked. However, the area for the element data of the reused structure is not reused.

In KL1 applications, it often happens that the areas of reclaimed structures can be reused for successive allocation. This is frequent in programs for list processing and programs written in message driven programming. In the sample codes in Section 4.6.3, element H of the passive-unified list [H|L] is used as element H of the new list [H|Y], and is read and written by the instructions marked with stars ("⋆"). However, if the MRB of the *passive-unified* list is not marked, element H can actually be used in the new list as is, and, therefore, *read* and *write* instructions can be eliminated.

Therefore, the following new optimized instructions are introduced:

reuse_list_with_elements	*Reg*, $[F_{car}\|F_{cdr}]$
reuse_vector_with_elements	*Arity*,*Reg*,$\{F_0,F_1,...,F_n\}$

These instructions do nothing when the MRB of the structure pointer on *Reg* is not marked. If marked, they allocate a new structure, copy specified elements on the structure referenced by *Reg* to the new structure, and put the pointer to the new structure onto *Reg*. Thus, reuse of data structures reduces the number of memory operations and, accordingly, keeps the size of the working set small.

Sample code is shown as follows:

- optimized intermediate code:

```
      ⋮
app_loop:
  wait                R1, sus_or_fail
  is_list             R1, next
  commit
  read                R1, cdr, R4       % L
  reuse_list_with_elements R1, [1|0]
  alloc_variable      R5                % Y
  write               R5, R1, cdr
  unify_bound_value   R1, R2
  move                R4, R1
  move                R5, R2
  execute_tro         app_loop
      ⋮
```

In this code, *reuse_list* and instructions marked with stars ("⋆") are replaced with the *reuse_list_with_elements* instruction. The second argument *[1|0]* specifies that the head element has to be copied if the MRB of the list pointer on *R1* is marked. If the MRB is not marked, it does nothing and is equal to *nop*. Therefore, only the following *write R5,R1,cdr* instruction can allocate the list structure [H|Y]; the instruction works like the *rplacd* function in LISP. Consequently, in this example, reuse optimization can save one *read* and one *write* instructions and is worth approximately two machine steps.

4.6.5 Clause Indexing

The KL1 language neither defines the testing order for the clause selection nor has the backtracking mechanism. Thus, to attain quick suspension detection and quick clause selection, the compiler can arrange the testing order of KL1 clauses; this is called clause indexing. At first,

the compiler collects the clauses which test the same variable, and compiles the clauses into shared instructions. Most of these work as test-and-branch instructions with branch labels occurring in the instruction codes. All guard parts of a predicate are, then, compiled into a tree structure of instructions.

Our KL1 programming experiences up to now have told as that a clause is infrequently selected according to the type of argument but is often selected according to the value. Further, even if multi-way switching of KL1-B instructions on data types is introduced, these KL1-B instructions are eventually implemented by a combination of native binary branch instructions, in general. Consequently, we decided that KL1-B does not provide a multi-way switching instruction on data types, but just binary-branch KL1-B instructions on a data type. Additionally, KL1-B provides a multi-way jump instruction on the value of an instantiated variable.

Two instructions are added for multi-way jump on a value:

switch_atom *Reg*, [{X_1,L_1},{X_2,L_2}, ... ,{X_n,L_n}]
switch_integer *Reg*, [{X_1,L_1},{X_2,L_2}, ... ,{X_n,L_n}]

Switch_atom is used for multi-way switching on an atom value, and *switch_integer* is used for multi-way switching on an integer value. They test the value on the register *Reg*, and if it is equal to the value X_i, a branch to the instruction specified by the label L_i occurs. Since the internal algorithm implementing these switching instructions is not defined in KL1-B, the translator to a native code may choose the most suitable method for switching.

The current KL1-B instruction set was designed under several assumptions in terms of KL1 programs. Thus, we have to investigate how correct our assumptions are and how effective our KL1-B instruction set is.

5 Conclusion

This paper discussed design and implementation issues of the KL1 language processor. PIM architecture differs from Multi-PSI architecture because of its loosely-coupled network with messages possibly overtaken, and because of its cluster structure (i.e. its shared-memory multiprocessor portion). These differences greatly influence the KL1 language processor and are essential to parallel and distributed implementation of the KL1 language. Several of the implementation issues focused on in this paper are more or less associated with these features. Our implementation is a solution to this situation. ICOT has been working on these implementation issues intensively for the past four years, since 1988.

In this paper, we began by making several assumptions and, then, tailored our implementation to them. The assumptions came from our experiences based on the Multi-PSI system. Thus, we have to evaluate our implementation, accumulate experiences on our system, and verify the appropriateness of the assumptions. Hence, we will be able to reflect our results in the KL1 language processor of the next generation. In this development cycle, the systematic design concept is effective, and the concept yields the high modularity of a language processor. It turns out to be easy to improve and highly testable.

Our KL1 language processor is presented on the PIM systems (*PIM/p*, *PIM/c*, *PIM/i*, *PIM/k*), which are being demonstrated at FGCS'92.

Acknowledgment

We would like to thank all ICOT researchers and company researchers who have been involved in the implementation of the KL1 language so far, especially, Dr. Atsuhiro Goto, Mr. Takayuki Nakagawa, and Mr. Masatoshi Sato. We also wish to thank the R&D members of Fujitsu Social Science Laboratory. Through their valuable contributions, we have achieved a practical KL1 language processor. Thanks also to Dr. Evan Tick of University of Oregon, for his great efforts in evaluating the parallel garbage collector with us. We would also like to thank Dr. Kazuhiro Fuchi, Director of ICOT Research Center, and Dr. Shunichi Uchida, Manager of Research Department ICOT, for giving us the opportunity to develop the KL1 language processor.

References

[Baker 1978] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4), 1978, pp.280–294.

[Bevan 1989] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. *Parallel Computing*, 9(2), 1989, pp.179–192.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System PIMOS. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 230–251.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. of the Fourth Int. Conf. on Logic Programming*, 1987, pp.276–293.

[Crammond 1988] J. A. Crammond. A Garbage Collection Algorithm for Shared Memory Parallel Processors. *Int. Journal of Parallel Programming*, 17(6), 1988, pp.497–522.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.208–229.

[Halstead 1985] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985, pp.501–538.

[Ichiyoshi *et al.* 1988] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *New Generation Computing*, Ohmsha Ltd. 1990, pp.159–177.

[ICOT 1st Res. Lab. 1991] ICOT 1st Research Laboratory. Tutorial on VPIM Implementation. *ICOT Technical Memorandum*, TM-1044, 1991 (*In Japanese*).

[Imai *et al.* 1991] A. Imai, K. Hirata and K. Taki. PIM Architecture and Implementations. In *Proc. of Fourth Franco Japanese Symposium*, ICOT, Rennes, France, 1991.

[Imai and Tick 1991] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *ICOT Technical Report*, TR-650, 1991. (To appear in IEEE Transactions on Parallel and Distributed Systems)

[Inamura *et al.* 1988] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. of the North American Conf. on Logic Programming*, 1989, pp. 907–921 (also *ICOT Technical Report*, TR-466, 1989).

[Kimura and Chikayama 1987] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proc. of Symposium on Logic Programming*, 1987, pp.468–477.

[Nakagawa *et al.* 1989] T. Nakagawa, A. Goto and T. Chikayama. Slit-Check Feature to Speed Up Interprocessor Software Interruption Handling. In *IPSJ SIG Reports*, 89-ARC-77-3, 1989 (*In Japanese*).

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. of the Sixth Int. Conf. on Logic Programming*, 1989, pages 436–451.

[Nishida *et al.* 1990] K. Nishida, Y. Kimura, A. Matsumoto and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *Proc. of the Seventh Int. Conf. on Logic Programming*, 1990, pages 83–95.

[Rokusawa *et al.* 1988] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proc. of the 1988 Int. Conf. on Parallel Processing*, Vol. 1 Architecture, 1988,pp.18–22.

[Rokusawa and Ichiyoshi 1992] K. Rokusawa and N. Ichiyoshi. A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting. In *Proc. of Sixth Int. Parallel Processing Symposium*, IEEE, 1992.

[Sato and Goto 1988] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proc. of IFIP Working Conf. on Parallel Processing*, 1988, pp. 305–318.

[Takagi and Nakase 1991] T. Takagi and A. Nakase, Evaluation of VPIM: A Distributed KL1 Implementation – Focusing on Inter-cluster Operations –, In *IPSJ SIG Reports*, 91-ARC-89-27, 1991 (*In Japanese*).

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Tick 1991] E. Tick. *Parallel Logic Programming.* Logic Programming, MIT Press, 1991.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, (33)6, 1990, pp.494–500.

[Warren 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Articial Intelligence Center, SRI, 1983.

[Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proc. of Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, 1987, pp.432–443.

Author Index

Vol.1 1– 460
Vol.2 461–1218