Palo Alto Research Centers

# Controlling Large Software Development In a Distributed Environment

Eric Emerson Schmidt

XEROX

# Controlling Large Software Development in a Distributed Environment

**Eric Emerson Schmidt**

**CSL-82-7** **December 1982** [P82-00039]

© Copyright Eric Emerson Schmidt 1982. All rights reserved.

XEROX

Xerox Corporation
Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304

**Controlling Large Software Development in a Distributed Environment**

Report number CSL-82-7.

This report reproduces a dissertation submitted to the University of California, Berkeley in partial fulfillment of the degree requirements for the Doctor of Philosophy in Computer Science.

The research in this dissertation was supported by Xerox Corporation.

## Abstract

Breaking a program up into modules is an important technique for managing the complexity of large systems. As the number of modules increases, the modules themselves need to be managed. Changing even a single module can be difficult. Compilation and loading are complicated. Saving the state of a program for others to build on is quite error-prone. The development of a large program as part of a multi-person project is even worse. This thesis presents solutions to these problems. We use new languages to describe the modules that comprise a system and tools that automate software development.

The first solution developed is a version control system of modest goals that has been used to maintain up to 450,000 lines of code over the past year. Users of this system list versions of files in *description files* (DF files) that are automatically maintained for the user. DF files may refer to other DF files when one software package depends on another. A working set of software that is saved in a safe location is called a *release*. The need for a *release process* was identified and an iterative algorithm that uses DF files to perform releases has been developed.

Based on experience with the DF system and the desire to automate the entire compile-edit-debug-release cycle, a second solution was developed in which the development cycle is controlled by the *System Modeller*. The modeller automatically manages the compilation, loading, and saving of new modules as they are produced. The user describes his software in a *system model* that lists the versions of files used, the information needed to compile the system, and the interconnections between the various modules. The modeller is connected to the editor and is notified when files are edited and new versions are created. To provide fast response, the modeller behaves like an incremental compiler: only those modules that change are analyzed and recompiled.

**CR categories:** D.1.1 **[Programming Techniques]**: Applicative (Functional) Programming; D.2.2 **[Software Engineering]**: Tools and Techniques - *Modules and Interfaces*; D.2.7 **[Software Engineering]**: Distribution and Maintenance - *Version Control*; D.2.9 **[Software Engineering]**: Management - *Software Configuration Management*; D.3.2 **[Programming Languages]**: Language Classifications - *Applicative Languages*.

**Key words and phrases:** Version control, release process, Cedar, Mesa, software management, automatic compilation, applicative language, modules, DF files, system models.

# TABLE OF CONTENTS

Chapter 1 — Page 1
Chapter 2 — Page 15
Chapter 3 — Page 49
Chapter 4 — Page 75
Chapter 5 — Page 103
References — Page 107
Appendix A — Page 111
Appendix B — Page 115
Appendix C — Page 125
Appendix D — Page 129
Appendix E — Page 135

## Acknowledgments

I would like to thank my committee, Robert S. Fabry and Richard J. Fateman of the Computer Science Department of U.C. Berkeley, James A. Reeds of the Statistics Department of Berkeley, and Butler W. Lampson of Xerox PARC's Computer Science Laboratory (CSL) for their comments on this thesis. Fabry was my principal research adviser throughout my years at Berkeley; his encouragement and support were constant.

Lampson is the intellectual father of the research described in this thesis. He coined the term "System Modelling" and designed the Cedar Kernel language, on which the SML language is based. Monthly meetings with him over the past three years gave me insight and encouragement in this research. Every idea in this thesis has been affected by his critical approach and incisive suggestions. Parts of section 3.4 are based on an internal memo by Lampson on the Cedar Kernel language.

Roy Levin of CSL had a strong influence on this research. He proposed the use of DF files as part of the Cedar release process, served as Release Master for almost all Cedar releases, and participated in the design of the Release Tool. His thorough approach is responsible for the success of the release process. He has contributed several ideas to the design, presented here, of a release process based on system models.

James H. Morris of CSL brought me into Xerox as a research intern five years ago. He remained my official sponsor at Xerox through another summer and later made me a member of the Cedar project, which he headed until this year. His strong personal influence through those years kept me directed toward a Ph.D.; he is both my friend and mentor.

System Modelling has been strongly influenced by Edwin H. Satterthwaite of CSL, who has used the system modeller heavily and has played a major role in providing fast turnaround through module replacement. Satterthwaite has given me advice on the design of the SM language and its implementation, and has allowed me to reproduce the system model that he developed for the Cedar compiler as an appendix to the thesis.

I would like to thank these fearless readers of earlier versions of my thesis: Mark Brown, Jim Donahue and Jim Horning of CSL, Brian Lewis of Xerox SDD, David Elliott of SRI International, William N. Joy of Sun Microsystems, Inc. and Dan Halbert of U.C. Berkeley. Each made valuable structural and syntactic improvements to this thesis. Lewis took the DF software described in Chapter 2, converted it to run in SDD's programming environment, and served as Release Master for their first release. Halbert carried many versions of this thesis between Palo Alto and Berkeley; his door-to-door service was invaluable.

Every member of the Cedar project has used some of the software described here. I thank them all for their patience in helping to debug these programs, and for their many suggestions that helped turn the programs into a usable system. Thanks also go to Paul Rovner and Warren Teitelman of CSL for allowing me to reproduce some of their DF files in the appendices to the

thesis. This research and the Cedar project would not have been possible without the support of CSL and particularly of its manager, Robert W. Taylor.

The text of this thesis was prepared using the Bravo editor, and the figures were done using the SIL illustrator. Both of these programs were run on a Dorado. These three tools made preparation of this thesis much easier.

I would like to thank my wife, Wendy, for editing this thesis as it was prepared and for supporting me through the good and bad times.

This thesis is dedicated to the memory of my father, Dr. Wilson Schmidt, Professor of Economics, who died while this thesis was being prepared. His emphasis on the importance of graduate education and the personal value of a Ph.D. was the inspiration for my pursuit of a Ph.D.

# 1. Introduction

## 1.1 Introduction

This dissertation presents a solution for problems occurring when large numbers of components of a software system are developed and maintained by many different programmers. Tools and languages are described that automate production of new versions of the system and the integration of packages into a stable system. The solution is developed in the context of a project to build Cedar, a new programming environment at Xerox's Palo Alto Research Center.

The Cedar project [Deutsch-Taft, 1980] is an attempt to take the Mesa language and build around it a programming environment based on ideas from Interlisp and Smalltalk, while retaining the strong type-checking properties of Mesa. Cedar makes it easy for programmers to share packages and collaborate on software development. Cedar programmers work in a distributed computing environment and have to be able to share each other's programs in various stages of development. In this setting, control of versions and file management is difficult because of the large number of files in Cedar and the requirement that versions of files must agree.

The first solution developed manages versions of files based on *description files* (DF files) that have information about versions of files needed and their locations. DF files that describe packages of software are input to a *release process*. The release process checks the submitted DF files to see if the programs they describe are made from compatible versions of software, and, if so, copies the files to a safe location. The *Release Tool* performs these checks and copies the files; an interactive algorithm is used that can be repeated after errors in DF files are found and fixed. Use of the Release Tool allows the person making the release, who is called the *Release Master*, to release software with which he is not familiar.

The DF system automates version control for the programmer. Based on experience with it, a second solution was developed that is a complete program management system. The user describes his software in *system models*, which are complete descriptions of a software system. Similar to a blueprint or schematic, a model combines in one place 1) information about the versions of files needed and hints about their locations, 2) additional information needed to compile the system, and 3) information about interconnections between modules, such as which procedures are used and where they are defined.

System models are manipulated by the *System Modeller*, a program that automates development of software in the Cedar programming environment. The system modeller is notified of new versions of files as they are created by the editor, and automatically recompiles and loads new versions of software. The modeller allows the user to maintain all three kinds of information, stored in system models that describe particular versions of a system. The system modeller is a complete software development system that will replace use of description files as the primary method of producing software in Cedar.

### 1.2 The Problem

Programs consisting of a large number of modules need to be managed. When the number of modules making up a system exceeds some small, manageable set, a programmer cannot be sure that every new version of each module in his program will be handled correctly. After each version is created, it must be compiled and loaded. The programmer may have to save it somewhere so others may use it. Without some automatic tool to help, the programmer cannot be sure that versions of software being transferred to another programmer are the ones intended.

A programmer unfamiliar with the composition of the program is more likely to make mistakes when a simple change is made. Giving this new programmer a list of the files involved is not sufficient, since he needs to know where they are stored and which versions are needed. A tool to verify a list of files, locations and versions is correct would help to allow the program to be built correctly. A program can be so large that simply verifying a description is not sufficient, since the description of the program is so large that it is impractical to maintain it by hand. An ideal support system would note new versions of modules and automatically manage the compilation, loading, and saving of new modules as they are produced.

The confusion of a single programmer becomes much worse, and the cost of mistakes much higher, when many programmers collaborate on a software project. In multi-person projects, changes to one part of a software system can have far-reaching effects. There is often confusion about the number of modules affected and how to rebuild affected pieces. For example, user-visible changes to heavily-used parts of an operating system are made very seldom and only at great cost, since other programs that depend on the old version of the operating system have to be changed to use the newer version. To change these programs, the "correct" versions of each have to be found, each has to be modified, tested, and the new versions installed with the new operating system. Changes of this type often have to made quickly because the new system may be useless until all components have been converted. Members of large software projects are unlikely to make such changes without some automatic support.

The software management problems faced by the Cedar programmer when he is developing software are made worse by the size of Cedar software, the number of references to modules that must agree in version, and the need for explicit file movement between computers. The Cedar system now has 447,000 lines of Cedar code, and approximately 2000 source and 2000 object files. Almost all object files refer to other object files by explicit version stamp. A program will not run until all references to an object file refer to the same version of that file. Cedar is too large to store all Cedar software on the file system of each programmer's machine, so each Cedar programmer has to explicitly retrieve the versions he needs to run his system.

This thesis deals with problems that fall into the realm of *Programming-in-the-Large*, where the unit of discourse is the module, instead of *Programming-in-the-Small*, where units include scalar variables, statements, expressions, and the like [DeRemer-Kron, 1976].

**1.3 The Approach**

To provide solutions to the above problems, we take the following approach:

1) Languages are provided in which the user can describe his system.
2) Tools are provided for the individual programmer that automate management of versions of his programs. These tools are used to acquire the desired versions of files, automatically recompile and load a program, save new versions of software for others to use, and provide useful information for other program analysis tools such as cross-reference programs.
3) In a large programming project, software is grouped together as a release when the versions are all compatible and the programs in the release run correctly. The languages and tools for the individual programmer are extended to include information about cross-package dependencies. The release process is designed so production of releases does not lower the productivity of programmers while the release is occurring.

To accomplish 1-3, we must identify the kinds of information that must be maintained to describe the software systems being developed. The information needed can be broken down into three categories:

1. File Information: For each version of a system, the versions of each file in the system must be specified. There must be a way of locating a copy of each version in a distributed environment. Because the software is always changing, the file information must be changeable to reflect new versions as they are created.
2. Compilation Information: All files needed to compile the system must be identified. It must be possible to compute which files need to be translated or compiled or loaded and which are already in machine runnable format. (This is called "Dependency Analysis.") The compilation information must also include other parameters of compilation such as compiler switches or flags that affect the operation of the compiler when it is run.
3. Interface Information: In languages that require explicit delineation of interconnections between modules (e.g., Mesa, Ada), we must be able to express these interconnections.

**1.4 Solutions**

The research for this thesis was done in two steps. The first system, based on DF files, gave important experience with the release process. A second system that included all the functionality of DF software plus an automatic program development system was then developed.

*DF Software and the Release Process*

A system with modest goals and low overhead was developed and has been in use for more than a year. Each programmer lists files that are part of his system in a *description file* (called a *DF* file). Each entry in a DF file consists of a file name, its location, and the version desired.

The programmer can use tools to retrieve files listed in a DF file and to save new versions of files in the location specified in the DF file. Because recompiling the files in his system can involve use of other systems, DF files can refer also to other DF files. The programmer can verify that, for each file in the DF file, the files it depends on are also listed in the DF file.

DF files are input to a *release process* that verifies that the cross-package references in DF files are valid. The dependencies of each file on other files are checked to make sure all files needed are also part of the release. The release process copies all files to a place where they cannot be erroneously destroyed or modified.

The information about file location and file versions in DF files is used by programs running in the Xerox environment. Each programmer has a personal computer on which he develops software. Each personal computer has its own disk and file system. Machines are connected to other machines using an Ethernet. Files can be transferred by explicit request from the file system on one machine to another machine. Often transfers are between a personal machine and a *file server*, which is a machine dedicated to servicing file requests.

The major research contributions of the DF system are 1) a language that, for each package or system described, differentiates between files that are part of the package or system and files needed from other packages or systems, and 2) a release process that does not place too high a burden on programmers and that pulls together packages being released. A release is *complete* if and only if every object file needed to compile every source file is among the files being released. A release is *consistent* if and only if only one version of each package is being released and every other package depends on the version being released. The release process is controlled by a person acting as a *Release Master*, who spends a few days per monthly release running programs that verify that the release is consistent and complete. Errors in DF files, such as references to non-existent files or references to the wrong versions of files, are detected by a program called the *Release Tool*. After errors are detected, the Release Master contacts the implementor and has him fix the DF file.

Releases can be frequent since performing each release imposes a low cost on the Release Master and on Cedar programmers. The Release Master does not need to know details about the packages being released, which is important when the software of the system becomes too large to be understood by any one programmer. Indeed, no single programmer has ever known how to rebuild the entire Cedar system. The implementor of each package can continue to make changes to his package until the release occurs, secure in the knowledge that his package will be verified before the release completes. Many programmers make such changes at the last minute before the release. The release process supports a high degree of parallel activity by programmers engaged in software development.

The DF system does not offer all that is needed to automate software development. DF files have only that information needed to control versions of files. Interconnections between Cedar modules that are loaded must be specified in another language called C/Mesa. No support for automatic recompilation of changed software modules is provided in the DF system; the only tool provided is a consistency checker that verifies that an existing system does not need to be

recompiled. DF files cannot be used as replacements for C/Mesa descriptions and cannot be used as input to a recompilation tool; there is not enough information in DF files to do this.

*Complete Solution: System Modelling*

A complete software system for Cedar has been built. It provides automatic support for the edit-compile-debug-release cycle. Its design is based in part on experience gained from the DF system and the release process.

This software management system uses information stored in *system models*, which contain the three kinds of information that must be managed. System models replace DF files and C/Mesa descriptions. The SML language, in which system models are written, allows complete descriptions of all interconnections between Cedar modules. Since these interconnections can be very complicated, the language includes defaulting rules that simplify system models in common cases.

The Cedar programmer uses the System Modeller to manipulate systems described by the system models. The system modeller 1) manipulates the versions of files listed in models, 2) tracks changes made by the programmer to files listed in the models, 3) automatically recompiles and loads the system, and 4) provides complete support for the release process. The modeller is connected to the Cedar editor and is notified when files are edited and new versions are created. The modeller recompiles new versions of modules and any modules that depend on them. To provide fast response, the modeller behaves like an incremental compiler: only those modules that change are analyzed and recompiled.

The main research contributions are 1) an extremely powerful module interconnection language that expresses interconnections as complicated as those found in Cedar, 2) a user interface that allows interactive use of the modeller while maintaining an accurate description of the system, and 3) the data structures and algorithms developed to maintain caches that enable fast analysis of modules by the modeller.

The tools that utilize the information above must 1) be usable by programmers working alone and as part of large projects; and 2) be suitable for a distributed environment. To achieve these goals, we have solved problems in a number of areas.

First, the software management tools presented here are easy to use and perform their functions quickly. The tools described are designed to run while the programmer is developing his software and automatically update system descriptions whenever possible. It is important that the tools be used while the programmer is developing software so he can get the most benefit from them. When components are changed, the descriptions are adjusted to refer to the changed components. Manual updates of descriptions by the programmer would slow his software development and proper voluntary use of the system seems unlikely. These tools function like an incremental compiler: only those pieces of the system that change are recompiled, loaded, and saved.

Second, the tools described here run in Xerox PARC's computing environment. At Xerox, each programmer has his own personal computer, which is connected to other computers over an Ethernet. This environment introduces two types of delays in access to versions of software stored in files: 1) if the file is on a remote machine, it has to be found, and 2) once found, it has to be retrieved. Since retrieval time is determined by the speed of file transfer across the network, we try to avoid retrieving files when the information we want about a file can be computed once and stored in a database. For example, the size of data needed to compute recompilation information about a module is small compared to the size of the module's object file. Recompilation information can be saved in a database stored in a file on the local disk for fast access. In cases where the file *must* be retrieved, determining which machine and directory has a copy of the version desired can be very time-consuming. The file servers can deliver information about versions of files in a directory at a rate of up to six versions per second. Since directories can have many hundreds of versions of files, it is not practical to enumerate the contents of a file server while looking for a particular version of a file. The solution presented here depends on the construction of databases for each software package or system that contains information about file locations.

(Other distributed computing environments many be organized to provide a *uniform file system* (UFS). Such a file system hides the fact that files may be stored on separate file systems and provides the illusion of one large name space. One might expect that future file systems will be UFS's and thus that we are partly solving problems that are artifacts of the current Xerox environment. On the contrary, we believe that there will always be cases where a UFS cannot be adopted. For example, a UFS needs to have available a high speed network to provide file access at speeds close to those of a file system on a disk attached directly to the local machine. Also, differences between the policies under which computers are managed may prevent the unification of file systems into a UFS.)

Third, since Cedar modules have a complicated interconnection structure, the system modeller includes a description language that can express the interconnection structure between Cedar modules. These interconnection structures are maintained automatically for the programmer. When new interconnections between modules are added by the programmer, the modeller updates the model to add the interconnection when possible. This means the user has to maintain these interconnections very seldom. The modeller checks interconnections listed in models for accuracy by checking the parameterization of modules. This checking of large numbers of modules is very time-consuming in the distributed environment discussed earlier, so the solutions adopted also involve use of local databases with such information.

## 1.5 Organization of this Thesis

The research described in this thesis is divided into three somewhat independent parts. Each part is given a chapter. The rest of this first chapter contains descriptions of other systems that deal with software management and version control. Subsequent chapters also contain references to these systems.

Chapter 2 describes the (now largely complete) system that uses DF files for managing versions of software that has been used in the Cedar project for about thirteen months. After a discussion of the Cedar environment, the programs run by individual programmers to manage versions of their own software are presented. The need for Cedar releases is explained, and the duties of the Release Master in charge of making the release are described. A list of the steps required for a release is followed by information about experience in the Cedar project with the release process. One appendix at the end of the thesis gives information on the size and frequency of Cedar releases during the past year. Another appendix gives examples of versions of software submitted to a recent Cedar release.

The principles underlying the SML language are presented in Chapter 3. This language is unusual because it is polymorphic (types are full-fledged values) and applicative (the language has no variables). The chapter reveals the fundamental structure of SML and documents the way systems of Cedar modules are described in SML. Issues of implementation are discussed along with plans for unification of SML and the Cedar language.

Chapter 4 describes the system modeller and shows how it uses system models as the description of a system to be compiled, loaded, run, and saved on a central file server. The connection between the Cedar editor and Modeller, which enables the modeller to track changes made by the programmer and automatically re-compile and load his system, is described. The basic edit-compile-debug-release cycle is presented, followed by details on implementation. The complexity of processing information about large systems is reduced by the use of specialized tables indexed by unique-ids. A mechanism for making Cedar releases is presented, followed by information on the existing implementation of the system modeller. One appendix contains the interface between the system modeller and the debugger, another shows a small model currently in use, and the last appendix is a copy of the largest model in use.

Chapter 5 consists of a conclusion and some suggestions for future research. References are at the end of Chapter 5.

## 1.6 Previous Work

There has been little research in version control and automatic software management. Of that, almost none has built on other research in the field. Despite good reasons for it (the many differences between program environments, and the fact that programming environments usually emphasize one or two programming languages, so the management systems available are often closely related to those programming languages), this fact reinforces the singularity of this

research. I, too, must build from first principles by virtue of two elements: 1) the description mechanism needed to describe Cedar module interconnections is more complex than that supported by other systems, and 2) no other research in this area has focused on the problems that occur when software development is done on separate computers without a common file system. Nevertheless, it is well worth reviewing previous work in this area.

### *Make*

The *Make* program uses a system description called the *Makefile*, which lists an acyclic dependency graph (explicitly given by the programmer). For each node in the dependency graph, the Makefile contains a *make rule*, which is to be executed to produce a new version of the parent node if any of the son nodes change [Feldman, 1979].

For example, the dependency graph in Figure 1.1 shows that x1.o depends on x1.c, and the file a.out depends on x1.o and x2.o.

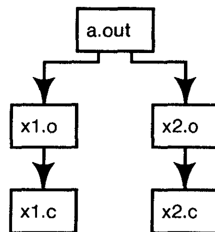



Figure 1.1

The Makefile that represents this graph is shown in Figure 1.2.

```
a.out: x1.o x2.o
        cc x1.o x2.o
x1.o: x1.c
        cc -c x1.c
x2.o: x2.c
        cc -c x2.c
```

Figure 1.2

Figure 1.2 gives "cc -c x1.c" as the command to execute to produce a new version of x1.o when x1.c is changed. *Make* decides to execute the make rule (i.e., compile x1.c) if the file modification time of x1.c is newer than that of x1.o.

The description mechanism shown in Figure 1.2 is intuitively easy to use and explain. The simple notion of dependency (e.g., a file x1.o that depends on x1.c must be recompiled if x1.c is newer) works correctly virtually all the time. The Makefile can also be used as a place to keep useful commands the programmer might want to execute, e.g.,

```
print:
        pr x1.c x2.c
```

defines a name "print" that depends on no other files (names). The command "make print" will print the source files x1.c and x2.c. There is usually only one Makefile per directory and, by convention, the software in that directory is described by the Makefile. This makes it easy to examine unfamiliar directories simply by reading the Makefile.

*Make* is an extremely fast and versatile tool that has become very popular among UNIX users. Unfortunately, *Make* uses modification times from the file system to tell which files need to be re-*made*. These times are easily changed by accident and are a very crude way of establishing consistency. Often the programmer omits some of the dependencies in the dependency graph, sometimes by choice. Thus, even if *Make* used a better algorithm to determine the consistency of a system, the Makefile could still omit many important files of a system.

### *SCCS*

The Source Code Control System (SCCS) manages versions of C source programs, enforcing a check-in and check-out regimen, controlling access to versions of programs being changed [Glasser, 1978], [Ivie, 1977], and [Rochkind, 1975].

A programmer who wants to change a file under SCCS control does so by 1) gaining exclusive access to the file by issuing a "get" command, 2) making his changes, and 3) saving his changed version as part of the SCCS-controlled file by issuing a "delta" command. His changes are called a "delta" and are identified by a release and level number (e.g., "2.3"). Subsequent users of this file can obtain a version with or without the changes made as part of delta 2.3. While the programmer has "checked-out" the file, no other programmers may store new deltas. Other programmers may obtain copies of the file for reading, however. SCCS requires that there be only one modification of a file at a time. There is much evidence this is a useful restriction in multi-person projects [Glasser, 1978].

SCCS stores all versions of a file in a special file that has a name prefixed by "s.". This "s." file represents these deltas as insertions, modifications, and deletions of lines in the file. Their representation allows the "get" command to be very fast.

### *SMF*

*Make* and SCCS were unified in special tools for a development project at Bell Labs called the *Software Manufacturing Facility* (SMF) [Cristofor, *et al.*, 1980]. The SMF used *Make* and SCCS augmented by special files called *slists*, which list desired versions of files by their SCCS version number.

An slist may refer to other slists as well as files. In the SMF, a system consists of a *master slist* and references to a set of slists that describe subsystems. Each subsystem may in turn describe other subsystems or files that are part of the system. The SMF introduces the notion of a

*consistent* software system: only one version of a file can be present in all slists that are part of the system. Part of the process of building a system is checking this consistency.

SMF also requires that each slist refer to at least one Makefile. Building a system involves 1) obtaining the SCCS versions of each file, as described in each slist, 2) performing the consistency check, 3) running the Make program on the version of the Makefile listed in the slist, and 4) moving files from this slist to an appropriate directory. Figure 1.3 shows an example of a hierarchy of slists, where ab.sl is the master slist.

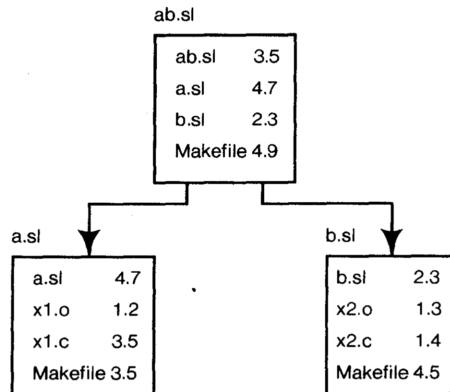



Figure 1.3

SMF includes a database of standard versions for common files such as the system library. Use of SMF solves the problem created when more than one programmer is making changes to the software of a system and no one knows exactly which files are included in the currently executing systems.

*PIE*

The PIE project is an extension to Smalltalk, implementing a network database of Smalltalk objects (i.e., data and procedures) and more powerful display and usage primitives. [See all of the Goldstein-Bobrow papers]. PIE allows users to categorize different versions of a Smalltalk object into *layers*, which are typically numbered starting at 0. A list of these layers, most-preferred layer first, is called a *context*. A context is a search path of layers, applied dynamically whenever an object in the network database is referenced. Among objects of the same name, the one with the layer number that occurs first in the context is picked for execution. Whenever the user wants to switch versions, he arranges his context so the desired layer occurs before any other layers that might apply to his object. The user's context is used whenever any object is referenced.

The distinction of PIE's solution to the version control problem is the ease with which it handles the display of and control over versions. PIE inserts objects (PIE procedures) into a network that corresponds to a traditional hierarchy plus the threads of layers through the network. The links of the network can be traversed in any order. As a result, sophisticated analysis tools can examine the logically-related procedures that are grouped together in what is called a

Smalltalk "class." More often, a PIE browser is used to move through the network. The browser displays the "categories" (a grouping of classes) in one corner of a window. Selection of a category displays a list of classes associated with that category, and so on until a list of procedures is displayed. By changing the value of a field labeled "Contexts:," the user can see a complete picture of the system as viewed from each context. This interactive browsing feature makes comparison of different versions of software very convenient.

*Gandalf*

The Gandalf project at CMU is implementing parts of an integrated software development environment for the GC language, an extension of the C language [Habermann, et al., 1982]. Included are a syntax-directed editor, a configuration database, and a language for describing what they call *system compositions* [Habermann-Perry, 1980] and [Habermann, 1979a]. Various Ph.D. theses have explored their language for system composition [Cooprider, 1979] and [Tichy, 1980].

Recent work on a System Version Control Environment (SVCE) combines Gandalf's system composition language with version control over multiple versions of the same component [Kaiser-Habermann, 1982]. Parallel versions, which are different implementations of the same specification, can be specified using the name of the specific version. There may be serial versions of each component, which are organized in a time-dependent manner. One of the serial versions (called a *revision*) may be referenced using an explicit time stamp. One of these revisions is designated as the "standard" version that is used when no version is specified.

Descriptions in the System Version Control Language (SVCL) specify which module versions and revisions to use. A *collection* of logically-related software modules is described by a *box* that names the versions and revisions of modules available. Boxes can include other boxes or modules. A module lists each parallel version and revision available. Other boxes or modules may refer to each version using postfix qualifiers on module names. For example, "M" denotes the standard version of the module whose name is "M," and "M.V1" denotes parallel version V1. Each serial revision can be specified with an "@," e.g., "M.V1@2" for revision 2.

Each of these expressions, called *path names*, identifies a specific parallel version and revision. Pathnames behave like those in the UNIX system: a path name that begins, for example, /A/B/C refers to box C contained in box B contained in A. Pathnames without a leading "/" are relative to the current module. *Implementations* can be used to specify the modules of a system, and *compositions* can be used to group implementations together and to specify which module to use when several modules provide the same facilities. These ways of specifying and grouping versions and revisions allow virtually any level of binding: the user may choose standard versions or, if it is important, he can be very specific about versions desired. The resulting system can be modified by use of components that specialize versions for any particular application. (See Figure 1.4).

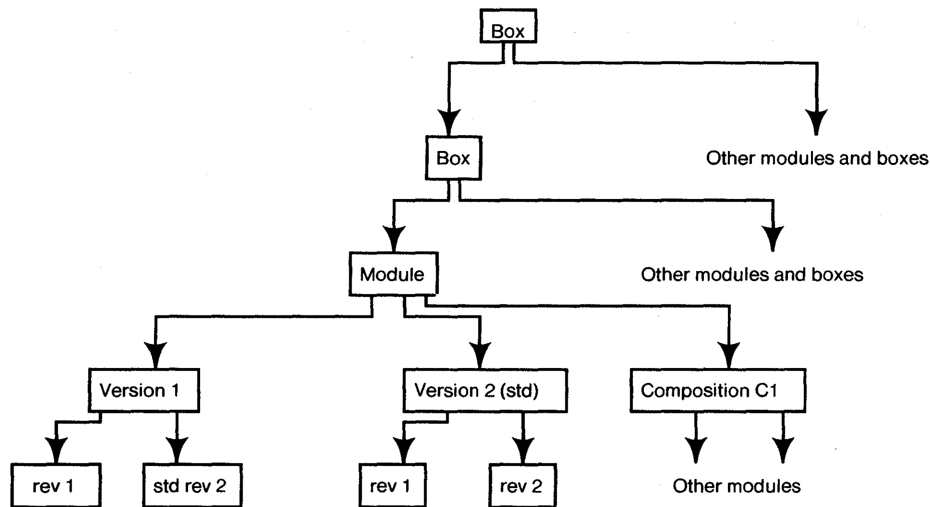



Figure 1.4

SVCE also contains facilities for "System Generation." The Gandalf environment provides a command to make a new *instantiation*, or executable system, from an implementation or composition. This command compiles, links, and loads the constituent modules. The Gandalf editor is used to edit modules and edit SVCL implementations directly, and the command to build a new instantiation is given while using the Gandalf editor. Since the editor has built-in templates for valid SVCL constructs, entering new implementations and compositions is very easy.

SVCE combines system descriptions with version control, coordinated with a database of programs. Of the existing systems, this system comes closest to fulfilling our three requirements described earlier: Their file information is in the database, their recompilation information is represented as links in the database between programs, and their interface information is represented by system compositions.

*Intermetrics Approach*

A system used to maintain a program of over one million lines of Pascal code is described in [Avakian, *et al.*, 1982]. The program is composed of 1500 separately-compiled components developed by over 200 technical people on an IBM 370 system. Separately-compiled Pascal modules communicate through a database (called a *compool)* of common symbols and their absolute addresses. Because of its large size (90 megabytes, 42,000 names), a compool is stored as a base tree of objects plus some incremental revisions. A simple consistency check can be applied by a link editor to determine that two modules were compiled with mutually-inconsistent compools, since references to code are stamped with the time after which the object file had to be recompiled.

Management of a project this size poses huge problems. Many of their problems were caused by the lack of facilities for separate compilation in standard Pascal, such as interface-implementation distinctions. (Better facilities are available in Mesa.) The compool includes all symbols (procedures and variables) that are referenced by modules other than the module in which they are declared. This giant interface between modules severely restricts changes that affect more than one separately-compiled module. Such a solution is only suitable in projects that are tightly managed. Their use of differential-updates to the compool and creation times to check consistency makes independent changes by programmers on different machines possible, since conflicts will ultimately be discovered by the link editor.

*Mesa, C/Mesa, and Cedar*

Because the research described in this thesis is implemented in Cedar, the reader has to be familiar with the existing Cedar/Mesa environment. The description below is repeated with different emphasis in Chapters 2 and 3.

Mesa programs can be one of two kinds: interfaces (or definitions) and implementations [Mitchell, *et al.*, 1979]. The code of a program is in the implementation, and the interface describes the procedures and types (as in Pascal) that are available to client programs. These clients reference the procedures in the implementation file by naming the interface and the procedure name, exactly like record or structure qualification (e.g., RunTime.GetMemory[] refers to the procedure GetMemory in the interface RunTime). The Mesa compiler checks the types of both the parameters and results of procedure calls so that the procedures in the interfaces are as strongly type-checked as local, private procedures appearing in a single module.

The interconnections are implemented using records of pointers to procedure bodies, called *interface records.* Each client is passed a pointer to an interface record and accesses the procedures in it by dereferencing once to get the procedure descriptors, which are an encoded representation sufficient to call the procedure bodies.

A connection must be made between implementations (or *exporters*) and clients (or *importers*) of interfaces. In Mesa this is done by writing programs in C/Mesa, a language that was designed to allow users to express the interconnection between modules, specifying which interfaces are exported to which importers. With sufficient analysis, C/Mesa can provide much of the information needed to recompile the system. However, C/Mesa gives no help with version control since no version information can appear in C/Mesa configurations.

Using this configuration language, users may express complex interconnections, which may possibly involve interfaces that have been renamed to achieve information hiding and flexibility of implementation. In practice, very few configuration descriptions are anything more than a list of implementation and client modules, whose interconnections are resolved using defaulting rules.

A program called the Mesa Binder takes object files and configuration descriptions and produces a single object file suitable for execution [Lauer-Satterthwaite, 1979]. Since specific versions of files cannot be listed in C/Mesa descriptions, the Binder tries to match the implementations

listed in the description with files of similar names on the invoker's disk. Each object file is given a 48-bit unique version stamp, and the imported interfaces of each module must agree in version stamp. If there is a version conflict (different versions of an interface), the Binder simply gives an error message and stops binding. Most users have elaborate command files to retrieve what they believe are suitable versions of files to their local disk.

A *Librarian* [Horsley-Lynch, 1979] is available to help control changes to software in multi-person projects. Files in a system under its control can be checked out by a programmer. While a file is checked out by one programmer, no one else is allowed to check it out until it has been checked in. While it is checked out, others may read it, but no one else may change it.

In one very large Mesa-language project [Harslem-Nelson, 1982], programmers submit modules to an *integration service* that recompiles all modules in a system quite frequently. A newly-compiled system is stored on a file system and testing begins. A team of programmers, whose only duty is to perform integrations of other programmer's software, fix incompatibilities between modules when possible. The major disadvantage of this approach is the amount of time between a change made by the programmer and when the change is tested.

The central problem with this environment is that even experienced programmers have trouble managing versions of Mesa or Cedar modules. The lack of a uniform file system, lack of tools to move version-consistent sets of modules between machines, and lack of complete descriptions of their systems contribute to the problem. This thesis addresses all these issues.

# 2. A Simple Version Control System

## 2.1 Introduction

We describe a system for efficient management of versions of software and its documentation. It is a system of modest goals and low overhead. This system is used to maintain the software of the Cedar project [Deutsch-Taft, 1980] in Xerox's Computer Science Laboratory. Cedar is a project to develop a prototype system that involves about 20 full-time implementors for about 20 users that consists of more than 4500 software files written in a high-level programming language.

*Goals*

The Cedar system changes frequently, both to introduce new function and also to fix bugs. Radical changes are possible and may involve recompilation of the entire system.

1. Our system must manage these frequent changes and must give us guarantees about the location and consistency of each set of files.
2. We want to call each consistent set of Cedar software a "Cedar Release," which must be a set of Cedar modules carefully packaged into a system that can be loaded and run on the programmer's personal machine. These releases must be carefully stored in one place, documented and easily accessible.
3. We want to be able to make Cedar releases as often as once a week, since frequent releases make available in a systematic way new features and bug fixes. The number of Cedar users is small enough that releases do not need to be bug-free since (in general) our users are tolerant of bugs in new components in the system. When bugs do occur, it must be clear who is responsible for the software in which the bug occurs.
4. Our scheme must minimize inconvenience to implementors and cannot require much effort from the person in charge of constructing the release. The scheme must not require a separate person whose sole job is control and maintenance of the system.
5. The system must be added on top of existing program development facilities, since we are unable to change key properties of our environment. In particular, we do not have a database system in which we can store information about versions of the system, and because of time constraints, we cannot alter the existing Cedar/Mesa programming environment in any significant ways. (Research for Cedar on databases, file systems, and program management described in Chapter 3 and 4 was in progress, but not ready, at the time this system was implemented.)

*Non-Goals*

We did not try to solve a number of common problems:

1. Because the system is undergoing rapid change, we are not concerned with long-term retention of versions of Cedar software, and plan to remove an old version as soon as a newer version is stable.
2. We assume each programmer submitting a package to a new release can adequately test his submission before the release, perhaps by running test programs or by making changes while running the previous release on his machine. If releases are easily made, fatal flaws are easily corrected by another release, e.g., the following day. We have not adopted a formal software testing strategy.
3. In our environment, each package has been implemented and is being maintained by one person, so we do not cope with multiple modifiers of the same component. See [Glasser, 1978], [Rochkind, 1975], [Horsley-Lynch, 1979], [Tichy, 1982] for descriptions of source code control, librarian checkin/checkout, and other schemes to handle this problem. Nor do we provide the branching or forking of versions of logically-related software available in SCCS.
4. In our system, new versions of files are complete copies of old versions. We do not need to store successive versions of a particular file in differential format (we view this as strictly a disk space - programmer time tradeoff.) We do not need to include and exclude groups of changes to get a particular version (in [Rochkind, 1975], they are called Deltas.)

In what follows, we describe the features of our environment that affected our approach, then describe our version control system as perceived by the individual user and programmer, and develop a working example. Then we describe how individual components are put together into a Cedar release. This is followed by sections on pragmatics and experience using this system.

## 2.2 Version Control in our Environment

Although we have developed a general system for management of files, we obtain additional consistency checking by using a limited understanding of the dependency relationships in the Cedar software systems. This dependency relationship is quite general, but to understand the specifics of our system we must give an overview of Cedar modules and dependencies.

*The Notion of Dependency*

A module A *depends on* another module B when a change to B may require a change to A. Throughout this paper we are concerned with the dependency of one module on another, since the system we describe manipulates versions of files. Nothing prevents adaptation of this scheme to dependency at a finer level.

If module A depends on module B, and B changes, then a system that contains the changed version of B and an unchanged version of A could be *inconsistent.* Depending on the severity of

the change to B, the resulting system may not work at all, or may work while being tested but fail after being distributed to users. Cedar requires inter-module version checking between A and B that is very similar to Pascal type-checking for variables and procedures. As in Pascal, Cedar's module version checking is designed to detect inconsistency as soon as possible at compile time so that the resulting system is more likely to run successfully after development is completed.

### *Dependency in Cedar*

Each Cedar module is represented as a source file (whose name ends in ".Mesa"). The Cedar compiler produces an object file (whose name ends in ".Bcd"). Each object file can be uniquely-identified by a 48-bit version stamp so no two object files have the same version stamp. Cedar modules depend on other modules (details given below) by listing in each object file the names and 48-bit version stamps of object files they depend on. Cedar requires that a collection of modules that depend on each other agree exactly in 48-bit version stamps. If module A depends on version 35268AADB3E4 (hexadecimal) of module B, but B has been changed and is now version 31258FAFBFE4, then the system is inconsistent.

The version stamp of a compiled module is a function of the source file and the version stamps of the object files on which it depends on. If module A depends on module B which in turn depends on module C, and C is changed and compiled, then when B and A are compiled their version stamps will change because of the change to C.

### *Types Of Dependency*

There are three kinds of modules in Cedar (interface, implementation, and configuration) and two programs (the Cedar compiler and binder) that produce object files.

Executing code for a Cedar system is contained in an implementation module. Each implementation module can contain procedures, global variables, and local variables that are scoped using Pascal scoping rules. To call a procedure defined in another implementation module, the caller (*client*) module must IMPORT a interface module that defines the procedure's type (the types of the procedure's argument and result values). This interface module must be EXPORTed by the implementation module that defines it (called the *implementor*).

Both the client and implementor modules depend on the interface module. If the interface is recompiled, both client and implementor must be recompiled. The client and implementor modules do not depend on each other, so if either is compiled the other does not need to be. Thus, Cedar uses the interface-implementor module distinction to provide type safety with minimal recompilation cost. This dependency is shown in Figure 2.1.

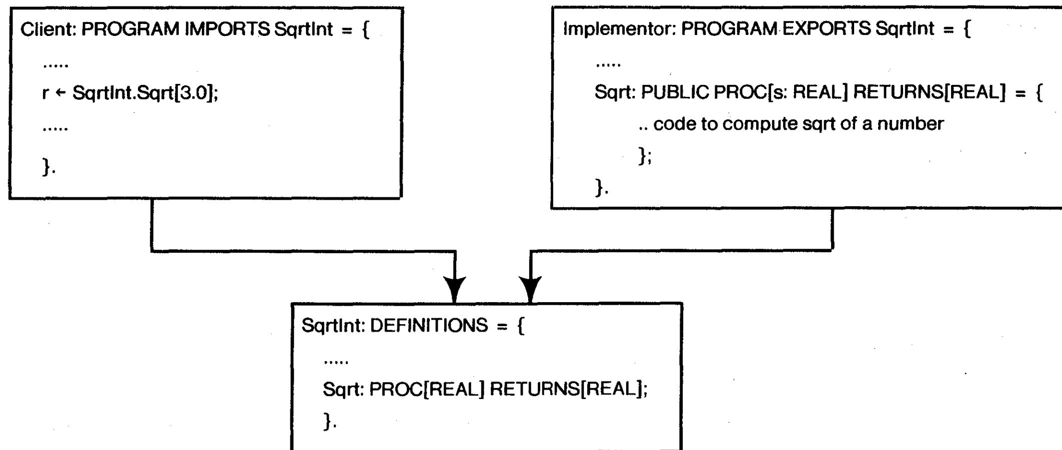



Figure 2.1 Importers and Exporters

A compiler-produced object file depends on 1) the source module that was compiled and 2) the object files of any interfaces that this module IMPORTS or EXPORTS. This dependency is shown in Figure 2.2. These interface modules are compiled separately from the implementations they describe, and interface object files contain explicit dependency information. In this respect, Cedar differs from most other languages with interface or header files (like C [Kernighan-Ritchie, 1978]). We will exploit this Cedar feature by analyzing the object files for interfaces produced by the Cedar compiler and use this to maintain our dependency information.

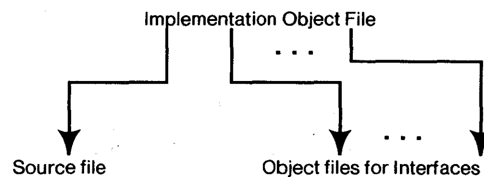



Figure 2.2

Another level of dependency is introduced by CONFIGURATION modules, which contain implementation modules or other configuration modules. The programmer describes a set of modules to be packaged together as a system by writing a description of those modules and the interconnections among them in a language called C/Mesa. A C/Mesa description is called a CONFIGURATION module. The source file for a configuration is input to the Cedar Binder which then produces an object file that contains all the implementation module object files. The Binder ensures the object file is composed of a logically-related set of modules whose imports and exports all agree in version. Large systems of modules are often made from a set of configurations (called sub-configurations). A configuration object file depends on 1) its source file and 2) the sub-configurations and implementation object files that are used to bind the configuration. These object files can be run by loading them with the Cedar loader which will resolve any IMPORTS not bound by the Binder.

In general, a Cedar system has a *dependency graph* that looks like Figure 2.3:

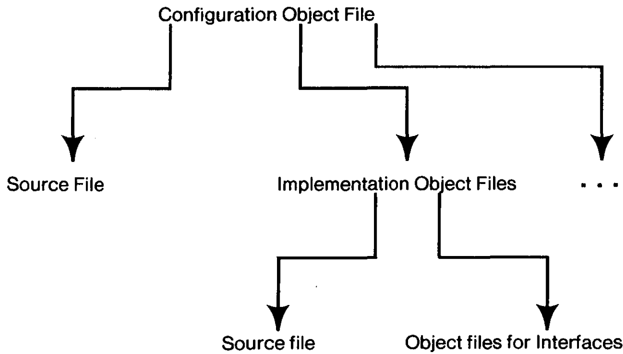



Figure 2.3

*Version And Size Problems*

Each Cedar programmer has his own personal computer, which is connected to other computers by an Ethernet. Most files comprising a system are stored on central file servers (machines dedicated to servicing file requests) and are copied from the central file server(s) to the personal machine by an explicit command (similar to the Arpanet "ftp" command). Figure 2.4 at the end of this chapter shows a typical environment. The owner of the machine must first install a *boot file* that is given control after the machine is powered on. Cedar users install the Cedar boot file that contains the Pilot operating system [Redell, *et al.*, 1979] and (possibly) pre-loaded programs.

Since the Binder and Loader ensure that the version stamps of Cedar modules all agree, all Cedar modules could be bound together and distributed to all users for use as the Cedar boot file. However, users who wanted to make changes would have to re-bind and load the system every time they changed a module to test their changes. The resulting boot file would be very large and difficult to transfer and store on the disks of the personal machines. To avoid these problems, Cedar users install this boot file on their machine (which contains a basic system to load and execute Cedar programs, a file system, and a pre-loaded editor) and then retrieve copies of programs they want to run that are not already in the boot file. These programs are loaded as they are needed.

Changes to these programs are possible as long as the versions of interfaces pre-loaded in the Cedar boot file agree with the versions IMPORTed by the program being loaded. Since the boot file EXPORTs more than 100 interfaces, the programmer can quickly become confused by version error messages for each of the interfaces he uses. This problem could be solved simply by disallowing changes to the Cedar interfaces (except, say, once a year). However, we want to be able to adjust interfaces frequently to reflect new features and refinements as they are understood.

### *Single vs. Shared File Systems*

The problem of determining which version of a module to use can occur on both single-user systems (as described previously) and also in conventional time-sharing systems with file systems shared among many users.

In a shared file system, different versions of files are usually stored in different sub-directories of the file system. The user must know which sub-directory to use when running his system. The most common error is a reference to a directory that is thought to contain a certain version of software, when in fact it contains a different version. Simply asserting that a directory contains a consistent, working version of software is not a sufficient guarantee of integrity, particularly when more than one programmer is involved and the normal file-system protections cannot distinguish among members of a privileged class of users. For example, the source programs for the UNIX kernel are often writeable by a user who has become a "super-user" by use of a magic password. Information about changes made by one super-user can be lost over time, which may lead to disastrous bugs in a system built from the resulting files. Use of SCCS to control changes helps as long as all modifiers obey the rules that require them to check out files before modifying them. The modifiers must not circumvent the normal SCCS procedures. However, even if each change is carefully recorded, there is no guarantee this version works with the rest of the system.

The use of a personal computer with its own file system in the Xerox environment has all the problems of timesharing systems and adds three extra problems:

1. The name space is flattened. Files from many directories and file servers are copied to the programmer's disk. This corresponds to copying all files that might be referenced on a shared file system into a single directory, which would be quite large. In fact, some Cedar users keep more than 1000 files on their personal machines!
2. The files used must be retrieved explicitly. Aside from the time required to transfer a large number of files, users can forget to copy every file and be left with a disk on which some of the files are consistent with one Cedar boot file and some are consistent with another.
3. Users generally develop software on their personal machines and then save copies on central file servers. When working on a large set of files, it is very easy to save some but not all of the files, leading to an inconsistent set of files on the central file servers. Should the contents of the local disk be destroyed, those unsaved versions would be lost. Other programmers, when told about software on the central file server, could retrieve an inconsistent system.

Each of these problems is addressed by a new Cedar File System under development. This file system was not available when the DF system was designed.

### *Value of Type Checking*

It is appropriate for us to pause and give some justifications for the strict type-checking rules in Cedar, especially those between modules.

Control of software in module interconnection languages is analogous to control over types in conventional programming languages, such as Pascal. Still opposed by some, strong type-checking in a language can be viewed as a conservative approach to programming, where extra rules (in the form of type equivalence) are imposed on the program. Proponents claim these rules lead to the discovery of many programming errors while the program is being compiled, rather than after it has started execution. Morris has said that having strong-type checking in a language is like using a "Neanderthal program verifier." [Morris, 1982]

Like strong type-checking of variables, type-checking in a language like Cedar with the explicit notion of an interface module can be performed at the module level so that incompatibilities between modules can be resolved when they are being collected together rather than when they are executing. As in the strong type-checking case, proponents claim this promotes the discovery of errors sooner in the development of programs.

Of course, incompatible versions of modules, like incompatible types in a programming language, must be corrected by the programmers involved. Many times, complex and subtle interdependencies exist between modules, especially when more than a few programmers are involved and the lines of communication between them are frayed or partially broken. In the Xerox environment, where each module is a separate file and development occurs on different personal machines, module-level type-checking is more important than type-checking of variables in conventional programming languages. This is because maintaining inter-module type consistency is by definition spread over different files, possibly on different computers by more than one programmer, while maintaining type-consistency of variables is usually localized in one file by one programmer on one machine.

### 2.3 A Single Component

In this section we describe procedures and programs used by an individual programmer to maintain his own software. The next section describes how these procedures are extended to manage Cedar releases.

We ask users to group logically-related files, such as the source and object files for a program they are developing, into a *package*. Each software package is described by a description file (called a *DF* file) that is a simple text file with little inherent structure that is editable by the programmer or user. The DF file lists all the files grouped together by the implementor as a package. For each file, the DF file gives a path name (or location) where the file can be found and information about which version is needed.

In Cedar, files are stored on file servers with names like "Ivy" and have path names (directory names) like "<Levin>BTrees>". A file like "BTreeDefs.Mesa" would be referenced as "[Ivy]<Levin>BTrees>BTreeDefs.Mesa". In addition, when created, each file is assigned a creation time. Therefore "BTreeDefs.Mesa Of May 13, 1982 2:30 PM" on "[Ivy]<Levin>BTrees>" defines a particular version.

A DF file is a list of such files. For syntactic grouping, we allow the user to list files grouped under common directories. The implementor of a B-tree package might write in his DF file, called BTrees.DF:

```
Directory [Ivy]<Levin>BTrees>
    BTreeDefs.Mesa          2-Oct-81 15:43:09
```

to refer to the file [Ivy]<Levin>BTrees>BTreeDefs.Mesa created at 2-Oct-81 15:43:09.

If, for example, the BTree package included an object file for BTreeDefs.Mesa, and an implementation of a B-tree package, it could be described in BTrees.DF as

```
Directory [Ivy]<Levin>BTrees>
    BTreeDefs.Mesa          2-Oct-81 15:43:09
    BTreeDefs.Bcd           2-Oct-81 16:00:28
    BTreeImpl.Mesa          2-Oct-81 15:28:54
    BTreeImpl.Bcd           2-Oct-81 16:44:31
```

Two different DF files could refer to different versions of the same file by using references to files with different create dates.

There are cases where the programmer wants the newest version of a file. If a ">" appears in place of a create time, the DF file refers to the newest version of a file on the directory listed in the DF file. For example,

```
Directory [Ivy]<Pilot>Defs>
    Space.Bcd               >
```

refers to the newest version of Space.Bcd on the directory [Ivy]<Pilot>Defs>. This is used mostly when the file is maintained by someone other than the programmer and he is content to accept the latest version of it.

*BringOver and StoreBack*

We encourage our users and implementors to think of the local disk on their personal machine as a cache of files whose "true" locations are the remote servers. We provide a command (called *BringOver*) that assures the versions listed in a DF file are on the local disk.

Since DF files are editable, the programmer who edits, for example, BTreeDefs.Mesa could, when ready to put a new copy on Ivy, store it manually and edit the DF file to insert the new create time for the new version.

For large numbers of files, this would always be error prone, so we provide a *StoreBack* command that provides automatic backup of changed versions (1) by storing files that are listed in the DF file but whose create date differs from the one listed in the DF (on the assumption that the file has been edited) and (2) by updating the DF file to list the new create dates. We also want the DF file to be saved on the file server, so we allow for a DF *self-reference* that indicates where the DF file is stored. For example, in BTrees.DF

Directory [Ivy]<Levin>BTrees>

| | | |
|---|---|---|
| BTrees.DF | 20-Oct-81 | 9:35:09 |
| BTreeDefs.Mesa | 2-Oct-81 | 15:43:09 |
| BTreeDefs.Bcd | 2-Oct-81 | 16:00:28 |
| BTreeImpl.Mesa | 2-Oct-81 | 15:28:54 |
| BTreeImpl.Bcd | 2-Oct-81 | 16:44:31 |

the first file listed is a self-reference. StoreBack arranges that the new version of BTrees.DF will have the current time as its create date.

*DF Imports*

The Cedar system itself is a set of implementation modules that export common system interfaces such as interfaces to the file system, memory allocator, and graphics packages. Assume the B-tree package uses an interface from the allocator. We require that users make this dependency explicit in their DF file. The BTree package will then IMPORT the interface "Space", which is stored in object form in the file "Space.Bcd".

The BTree DF package will reflect this dependency by "importing" Space.Bcd from a DF file "PilotInterfaces.DF" that lists all such interfaces. BTrees.DF will have an entry

Imports [Indigo]<Cedar>Top>PilotInterfaces.DF Of 2-Oct-81 15:43:09
Using[Space.Bcd]

The "Imports" in a DF file is analogous to the IMPORTS in a Cedar program. As in Cedar modules, BTrees.DF depends on Pilot.DF. Should "Space.Bcd" and its containing DF file "Pilot.DF" change, then BTrees.DF may have to change also.

The programmer may want to list special programs, such as a compiler-compiler or other preprocessors, that are needed to make changes to his system. This is accomplished using the same technique of Importing the program's DF file.

For the individual programmer, there are two direct benefits from making dependency information explicit in his DF file. 1) BringOver will ensure that the correct version of any imported DF files are on the local disk, so programmers can move from one personal machine to another and guarantee they will have the correct version of any interfaces they reference. 2) Listing dependency information in the DF file puts in one place information that is otherwise scattered across modules in the system.

*VerifyDF*

How does the programmer know which files to list in his DF file? For large systems, under constant development, the list of files is long and changes frequently. (Some packages depend on more than fifty other interfaces.) The programmer can run a program *VerifyDF* that analyzes the files listed in the DF file and warns about files that are omitted. VerifyDF analyzes the dependency graph described in section 2.2 and analyzes the versions of (1) the source file that was compiled to produce this object file and (2) all object files that this object file depends on. VerifyDF analyzes the modules listed in the DF file and constructs a dependency graph.

VerifyDF stops its analysis when it reaches a module defined in another package that is referenced by Imports in the DF. Any modules defined in other packages are checked for version-stamp equality, but no modules they depend on are analyzed, and their sources do not need to be listed in the package's DF file.

VerifyDF understands the file format of object files and uses it to discover the dependency graph, but otherwise it is quite general. For example, it does not differentiate between interface and implementation files. VerifyDF could be modified to understand object files produced by other language compilers as long as they record all dependencies in the object file with a unique version stamp. For each new such language, VerifyDF needs 1) a procedure that returns the object version stamp, source file name and source create time, and 2) a procedure that returns a list of object file names and object version stamps that a particular object file depends on.

If the programmer lists all such package and files he depends on, then some other programmer on another machine will be able to retrieve (using BringOver) all the files he needs to make a change to the program and then run StoreBack to store new versions and produce a new DF file.

Using these tools (BringOver, StoreBack, VerifyDF) the programmer can be sure he has a DF file that lists all the files that are needed to compile the package (*completeness*) and that the object files were produced from the source files listed in the DF file, and there are no version stamp discrepancies (*consistency*). The programmer can be sure the files are stored on central file servers and can turn responsibility for a package over to another programmer by simply giving the name of the DF file.

The reader may ask: given that a DF file describes a list of files that comprise a package or subsystem, why not use a subdirectory of a conventional tree-structured file system instead of a DF file? For a single programmer, DF files do not offer much of an advantage over such a tree-structured file system. The programmer could store all logically-related files in a common directory, and there would be a program analogous to VerifyDF that analyzes all files in a particular directory, rather than a DF file, for inconsistency. DF files offer more than directories offer when they are used for describing systems composed of specific versions (such as those in a directory) and also composed of references to other DF files without version information, such as "Imports" without specific versions. This is shown in the next section.

### 2.4 Releases

The previous section described how an individual programmer uses DF files to organize his software. We now describe how, with a few extensions, these same DF files can be used to describe releases of software. Releases are made by following a set of *Release Procedures*, which are essentially managerial functions by a *Release Master* and requirements placed on implementors. A crucial element of these Release Procedures is a program called the *Release Tool*, which is used to verify that the release is consistent and complete, and is used to move the

files being released to a common directory.

*What is a Release?*

If the packages a programmer depends on change very seldom, then use of the tools outlined in the previous section is sufficient to manage versions of software. However, at this stage in the Cedar project, packages that almost everyone depends on may be changed. A release must consist of packages that, for example, all use the same versions of interfaces supplied by others. If version mismatches are present, modules that IMPORT and EXPORT different versions of the same interface will not be connected properly by the loader. In addition to the need for consistency and completeness across an entire release (just as consistency and completeness were needed by individual packages), the component files of a particular release must be carefully saved somewhere where they are readily available and will not be changed or deleted by mistake, until an entire release is no longer needed.

*The Release Process*

We organize the administration of Cedar releases around an implementor who is appointed Release Master. In addition to running the programs that produce a release, he is expected to have a general understanding of the system, to make decisions about when to try to make a release, and to compose a message describing the major changes to components of the release.

Once he decides to begin the release process (after conferring with other implementors and users), the Release Master sends a "call for submissions" message through the electronic mail system to a distribution list of programmers who have been or are planning to contribute packages to the release. Over a period of a few days, implementors are expected to wait until new versions of any packages they depend on are announced, produce a new version on some file server and directory of their choosing, and then announce the availability of their own packages.

One message is sent per package, containing, for example, "New Version of *Pkg* can be found on [Ivy]<Schmidt>Pkg.DF, that fixes the bug ...". Programmers who depend on Pkg.DF are expected to edit their DF files (they would refer to Pkg.DF by an Imports clause) by changing them to refer to the new version. Since often it is the newest version, clients of Pkg.DF usually replace an explicit date by the ">" (greater than) symbol. They might refer to Pkg.DF by inserting

```
Imports [Ivy]<Schmidt>Pkg.DF Of >
    Using[File1.Bcd, File2.Bcd]
```

in their DF file.

If their package is not changing, they are expected to send a message to that effect. These submissions do not appear in lock step since changes by one implementor may affect packages that are "above" them in the dependency graph.

This pre-release integration period is a parallel exploration of the dependency graph of Cedar software by its implementors. If an implementor is unsure whether he will have to make changes as a result of lower level bug fixes, for instance, he is expected to contact the implementor of the lower package and coordinate with him. Circular DF-dependencies may occur, where two or more packages use interfaces exported by each other. In circular cases, the DF files in the cycle have to be announced at the same time or one of the DF files has to be split into two parts: a bottom half that the other DF file depends on and a top half that depends on the other DF file.

The Release Master simply monitors this integration process and when the final packages are ready, begins the release.

*The Release Tool*

Once all packages that will be submitted to the release are ready, the Release Master prepares a top-level DF file that lists all the DF files that are part of the release. Packages that are not changing (e.g., from a previous release) are also listed in this DF file. DF files are described using a construct similar to "Imports" discussed earlier. The contents of each DF file are referenced by an Include statement, e.g.,

Include [Ivy]<Levin>BTrees>BTrees.DF Of >

refers to the newest version of the BTree package stored on Levin's working directory <Levin>BTrees>. Include is treated as macro-substitution, where the entire contents of BTrees.DF are analyzed by the Release Tool as if they were listed directly in the top-level DF.

*Phases One and Two*

The Release Master uses the top-level DF as input to phase *one* (out of three) of the Release Tool. Phase one reads all the Included DF files of the release and performs a system-wide consistency check. A warning message is given if there are files that are part of the release with the same name and different creation times (e.g., BTreeDefs.Mesa of 20-May-82 15:58:23 and also another version of 17-Apr-82 12:68:33). Such conflicts may indicate that two programmers are using different versions of the same interface in a way that would not otherwise be detected until both programs were loaded on the same machine. These warnings may be ignored in cases where the Release Master is convinced that no harm will come from the mismatch. (For example, there is more than one version of Queue.Mesa in the Cedar release since more than one package has a queue implementation, but each version is carefully separated and the versions do not conflict.)

Phase one also checks for common blunders, such as a DF file that does not refer to newest versions of DF files it depends on, or a DF file that refers to Cedar files that do not exist where the DF file indicates they can be found. The Release Master makes a list, package by package, of such blunders and calls each person and notifies them they must fix their DF files.

Phase one is usually repeated once or twice until all such problems are fixed and any other warnings are judged benign. Phase *two* guarantees system wide completeness of a release by running VerifyDF on each component of the release. VerifyDF will warn of files that should

have been listed in the DF file but were omitted. Implementors are expected to run VerifyDF themselves, but during every release, someone forgets. Any omissions must be fixed by the implementor.

Once phases one and two are completed successfully, the Release Master is fairly certain there are no outstanding version or system composition problems, and he can proceed to phase three.

*Phase Three*

To have control over the deletion of old releases, phase three moves all files that are part of a release to a directory that is mutable only by the Release Master. Moving files that are part of the release also helps users by centralizing the files in one place. The DF files produced by implementors, however, refer to the files on their working directories. We therefore require that every file mentioned in the DF files that are being released have an additional phrase "ReleaseAs *releasePlace*". Our BTrees.DF example would look like

```
Directory [Ivy]<Levin>BTrees>
  ReleaseAs [Indigo]<Cedar>Top>
    BTrees.DF                20-Oct-81   9:35:09

Directory [Ivy]<Levin>BTrees>
  ReleaseAs [Indigo]<Cedar>BTrees>
    BTreeDefs.Mesa           2-Oct-81  15:43:09
    BTreeDefs.Bcd            2-Oct-81  16:00:28
    BTreeImpl.Mesa           2-Oct-81  15:28:54
    BTreeImpl.Bcd            2-Oct-81  16:44:31
```

which indicates a working directory as before and a place to put the stable, released versions. By convention, all such files must be released onto subdirectories of [Indigo]<Cedar>. To make searching for released DF files on the <Cedar> directory easier, each DF file's self-reference must release the DF file to the special subdirectory <Cedar>Top>. When the third phase is run, each file is copied to the release directory (e.g., B-tree files are copied to <Cedar>BTrees>) and new DF files are written that describe these files in their release positions, e.g.,

```
Directory [Indigo]<Cedar>Top>
  CameFrom [Ivy]<Levin>BTrees>
    BTrees.DF                9-Nov-81   10:32:45

Directory [Indigo]<Cedar>BTrees>
  CameFrom [Ivy]<Levin>BTrees>
    BTreeDefs.Mesa           2-Oct-81  15:43:09
    BTreeDefs.Bcd            2-Oct-81  16:00:28
    BTreeImpl.Mesa           2-Oct-81  15:28:54
    BTreeImpl.Bcd            2-Oct-81  16:44:31
```

The additional phrase "CameFrom" is inserted as a comment saying where the file(s) were copied

from.

The other major function of phase three is to convert references using the "newest version" notation (">") to be explicit dates, since "newest version" will change for every release. Phase three arranges that a reference like

Imports [Ivy]<Levin>BTrees>BTrees.DF Of >
Using[BTreeDefs.Bcd]

becomes

Imports [Indigo]<Cedar>BTrees>BTrees.DF Of *date*
CameFrom [Ivy]<Levin>BTrees>
Using[BTreeDefs.Bcd]

where *date* is approximately the time phase three is run.

Appendix B has more information, including an example of a DF file that was submitted to a release.

Figure 2.5 shows the steps taken to make a release.

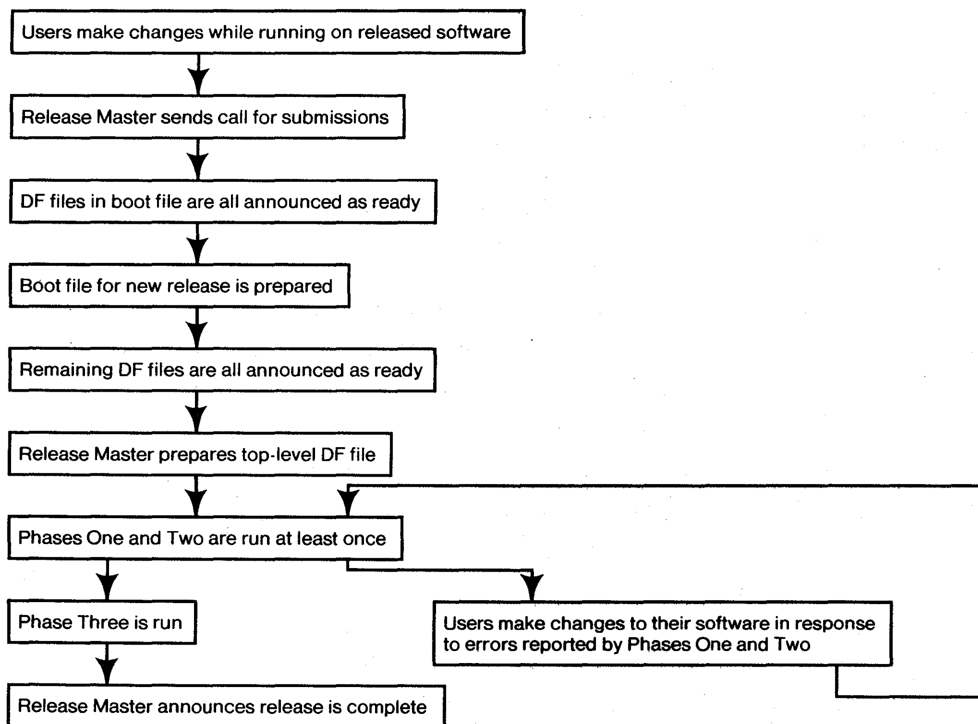



Figure 2.5 **Steps to Make a Release**

*Merits of This Approach*

The notion of a "Cedar Release" has many advantages. In addition to a strong guarantee that the software will work as documented, it has an important psychological benefit to users and implementors alike as a firewall against disasters, since programmers are free to make major changes that may not work at all, and are secure in the knowledge the last release is still available to fall back on. Since users can convert back and forth between releases, users have more control over which versions they use. There is nothing wrong with more than one such release being in use at one time by different programmers, since each programmer has his own personal machine. Users are allowed to convert to new Cedar releases at their own pace.

Our approach to performing releases fulfilled our initial goals:

1. All files in the release have been moved to the release directory. These files are mutually consistent versions of software. All DF files refer to files known to be on the release directory.
2. As described in section 2.2, we cannot make a configuration module that contains all the modules in a release. Cedar releases are composed of a) a boot file and b) programs that are mutually consistent and can be run on a personal machine with the boot file being released. Phase two runs VerifyDF on all the components to guarantee that the versions of source and object files listed in the DF file are the ones actually used to build the component and guarantees that all files needed to build the component are listed in the DF file, so no files that conflict in version can be omitted.
3. The release process is automatic enough that frequent releases are possible. Bugs in frequent releases are easily reported since the concept of ownership is very strongly enforced by our approach: The programmer who provides new versions of software is the recipient of bug reports of his software. Programmers who are out of town are expected to hand responsibility for their packages to another Cedar implementor while they are away.
4. The Release Master is required to a) decide when to make a release, b) send a call-for-submissions message, c) make a top-level DF file and run the Release Tool, and d) send a message announcing the release's completion. Because Cedar releases are expected, over time, to include more and more Cedar programs, it is important that the Release Master not need to compile packages other than any packages he may be contributing to the release. Indeed, no single person has ever known how to compile the entire Cedar system by himself.

   Since the implementors use DF files for maintaining their own software as well as for submitting components to the release, there is little additional burden on the implementors when doing a release. If the burden were too high, the implementors would delay releases and overall progress would be slowed as the feedback from users to implementors suffered.
5. We do not need a general database system to describe the dependency hierarchy of packages when we are producing systems. (In fact, in the first nine months of use, we did not have a complete list of dependencies in an easily accessible form.) For example, we use the message system, rather than a database of information that the programmers can query, to notify

implementors that packages they may depend on are ready. Each implementor is expected to understand the effect of his changes on his clients (i.e., the packages that depend on him).

It is possible to put an explicit date on the Include statement. Normally the newest version is wanted, but in situations where the implementor cannot wait until the release is complete, use of the explicit date guarantees the version intended by the programmer is the one used in the release and not a newer one. This would enable the parallel integrations defined in [Harslem-Nelson, 1982] where programmers submit their changes to an integration service and then continue to make more changes to their software while waiting to test their previously-submitted changes.

### 2.5 Pragmatics of Releases

In building this system we had to deal with several practical problems. Here are some of these problems and the pragmatic solutions we adopted.

*Includes and the Spanning Tree*

Programmers frequently agree to share a common package of files but are not sure who "owns" the package. We allow a DF file to "Include" another DF file when both DF files are being developed by the same programmer and are logically part of the same package. Phase one checks to make sure that, for each DF file, either 1) it is Included by the top-level DF file, or 2) it is indirectly Included by another DF file that is Included by the top-level DF file. This insures every DF file appears as an Include at least once. Phase one also checks that each DF file appears, at most, once and that this inclusion relationship is a spanning tree of DF files.

*Release Envelope and Exception List*

Some modules of Cedar were written by another division within Xerox and have not been changed by Cedar implementors. (Examples include the Pilot Operating System on which Cedar runs, and low-level disk utilities.) Programmers occasionally have to refer to these modules, but they are not being released or maintained. We consider these files to be outside the "release envelope," which contains the files we want to save as part of the release. Files that do not change and do not need to be under control of the release process are entered into an *exception list.* When phase one checks to make sure every file mentioned in the DF files, i.e., by Imports, is part of a package that is being released, files that appear in the exception list are not listed. A warning is given for all files that are not being released onto [Indigo]<Cedar>, since those files might be deleted or changed. No warning is given for files on the exception list.

*Deletion of Old Releases*

We have never needed more than two releases on the <Cedar> directory, so we run a program to delete an old release after a new release is safely stored on <Cedar>. This program is

given a list of top-level DF files for releases being kept and reads their contents, following the Include constraints to build a list of every file in the release(s) and then deletes any file on the <Cedar> directory that is not on the list of files being kept.

*Copying Files To The Release Directory*

The Release Tool runs on a personal machine and copies files from remote directory to remote directory, without making a copy of each file on the personal machine. This allows the Release Master to use the currently released version of Cedar to make a release of a new, incompatible version of Cedar. If the files being released were copied to the local disk, they would overwrite the older versions.

*Non-Program Files*

We require that every package include references to its documentation and command files to recompile the software. By convention, the documentation is always released to the <Cedar>Documentation> directory, making browsing of system documentation easy and guaranteeing the automatic deletion of old documentation along with old released software.

*Quality of Consistency Check*

Our check for the consistency of a release is based on creation times of files and 48-bit version stamps of Cedar interfaces. Programs or packages may not work together even if their mutual version stamps agree. The Release Tool's checks are not substitutes for testing of program changes prior to a release. Also, no checking is done on files that are part of the release but are not Cedar source and object files. For example, the wrong version of documentation files and command files to compile packages may be mentioned in DF files being released and we will not be able to check this. (This is discussed in more depth in Chapter 4.)

*Create Times as Unique IDs*

Unfortunately the create time listed in the DF file is not a unique-id since two programmers working on different computers could create a file with the same name at the same second. In practice this has never been a problem. Since the create time is only changed when the file is edited (i.e., when the contents change) this create time is preserved even when a file is copied or renamed. Phase one of the Release Tool confirms that all the files being released are consistent by comparing create times instead of version stamps, since the create times of a file can be found faster that its version stamp. In our experience, the create time is as good an indicator of consistency as the version stamp.

*Components Being Re-Released*

When preparing for a new release, we ask that implementors edit the DF files that are on the release directory instead of using the copy of the DF file they submitted to the last release. Although this means they have to switch the CameFrom statements back to ReleaseAs statements (we provide a tool to do this), it avoids the common problem of new release submissions Importing DF files that were released from working directories in the last release but are not being released in the next one.

*StoreBack and Files Missing*

A DF file may be submitted to the release that refers to a file not actually present on the working directory. This error can occur since StoreBack decides to store new versions of files on a working directory only if the version listed in the DF file is different. If the create time of the file on the local disk agrees with the create time of the entry in the DF file, no checking is normally done to see if the file is actually present on the working directory, since the check takes too long in normal development. To avoid this error, we now require that implementors run StoreBack with a special option that forces it to check that the file exists on the remote server even if the create time listed in the DF file and that of the file on the local disk agree.

*The Bootstrapping Problem*

Some changes, e.g., redesign of the garbage collector in Cedar or a new version of the Cedar instruction set emulated in microcode, present the most difficult integration problems. Inherent in these problems is the need for a cross-development system that allows editing and compiling of programs in an existing, working world to produce the new world to test. In small systems, one programmer can manage these changes by doing all cross-development himself and staging the changes to various levels of the software as they are debugged. Cedar is large enough that this is now impractical, and concurrent cross-development must be possible. For example, consider changes to the storage allocator and also to the window manager/editor of Cedar. The storage allocator must be tested with a program that uses the terminal for output, and the changes to the window manager must be tested using the storage allocator.

To permit concurrent development, each must use stable versions of the other's software, but such development usually requires interface changes that, at the very least, force recompilation of the other package and may involve more extensive edits to use the new interfaces. DF files encourage such concurrent development by simplifying the file movement that is part of the development of a new system and by allowing changes needed in order to let the other person to test his changes. For example, when the implementor of the storage manager needs to test his changes, he asks the implementor of the window manager 1) to save his new version of the window manager, 2) BringOver the released (stable) version of the window manager, 3) make changes to allow the storage changes to be tested, 4) run StoreBack to save those somewhere, and 5) BringOver the version saved in step (1) and continue his development. DF files do not

automate the bootstrap cycle itself since the two programmers must agree to the scheduling of this development, but they do allow the rapid movement between multiple versions of software in development.

**2.6 Experience**

We have had thirteen Cedar releases in eight months. Eleven of the releases have been "major releases" where many of the components of the release are changing, or, at the very least, are being recompiled. The other two were "mini releases" that were done a day after a major release to fix one or more catastrophic bugs that were in the major release.

There have been approximately four weeks between major releases. We can divide this period into three distinct time periods: *development*, *integration*, and *release*. The development phase is about three weeks long and is a time of unconstrained program development. New function is added and bugs in the previous release are fixed by the implementor during this time. A three to four day period of integration follows during which implementors who must coexist in the running system work out any bugs that surface when more than one package has changed. Finally, the phases of the release tool are run over a period of one or two days. Bugs in programs are less likely at this point. More often trivial DF file errors (wrong directory for an Imported DF file, etc.) are discovered.

We designed the release procedures so we could perform a Cedar release once a week. We have settled into a monthly cycle for two reasons: 1) user resistance to too-frequent changes and 2) the use of releases as planning targets by the project management.

1. Although we have a small user community, many users do not have the time to convert programs (and the system running on their personal computers) to new releases every week. In addition, each new release has to be coherently documented, and our users have to read the documentation. We have decided it is easier to deliver a well-debugged system once a month than a less well-debugged system once a week.
2. The Cedar project is managed using targets for the next and subsequent releases. These targets are informal and include plans to have certain new facilities in the next release. Since extensive changes take more than a week to design and implement, the project management cannot know whether a major change will be ready within a few weeks. Work proceeds on the assumption it will be ready. Concurrently, other implementors have made their changes and want to see them released. If the changes take longer than expected, a decision can then be made to proceed without the new component. As a result, monthly cycles have occurred naturally. This scenario occurred recently when the Cedar screen management facilities were going to replace those we had been using for a year. This change was so significant we planned to call it Cedar version 3.0 rather than increment the least significant digit of Cedar 2.5. Progress on the change slowed and bugs in the parts of the 2.5 release had to be fixed. We did a release of a version of Cedar called 2.6, with far fewer changes than were expected

in the release following 2.5 and then released Cedar 3.0 successfully five weeks later.

Is a weekly Cedar release attainable? We believe the answer is yes, since more frequent releases would allow less development time and thus fewer components would change per release, lowering integration and release time substantially. We imagine devoting four days of development, a morning of integration and afternoon to do a release. Such frequent releases would certainly involve many fewer people than our current monthly releases, and implementors who are not submitting new versions to a release will be able to continue development while the release is performed. However, the need for more frequent releases of Cedar has not been demonstrated so far.

## 2.7 Performance and Evolution

*Cedar*

The first automated Cedar release (Cedar 2.0 of October 1981) consisted of 1800 files and used 12 megabytes of disk storage. The size of Cedar releases has grown in a linear fashion to the largest (Cedar 3.5.2 of December 1982) of 4843 files, 63.5 megabytes, and 457,000 lines of Cedar source code. This growth is due to improvements to components that are re-released every release and also to the addition of packages developed independently and now ready for more extensive use. An example of the latter is the Cedar database package that has been in use for a number of years but is now being integrated into many tools in the Cedar environment.

The 3.5.2 version of Cedar was composed of 160 DF files, which were used to describe 79 release components. There is normally a one-to-one relationship between DF files and packages. Some components, however, are so large they are internally broken into five to ten DF files that are "Included" by an "umbrella" DF file.

The time to run the phases of the release varies greatly depending on how many and whose components are changing, and how many files have to be transferred. In a typical major release 50% of the components will be changing (for the 3.0 release it was 75%). Out of those components that do change, roughly 20% of the files in those components have not changed. The Release Tool does not analyze components that are not changing and does not store multiple copies of the same file. For a major release, phases one and two of the Release Tool are run for about an hour each, and phase three (which transfers the files) takes between three and six hours. These times are completely determined by the speed of and load on our central file servers, which are Xerox Alto personal computers that transfer files comparatively slowly. [Thacker, *et al.*, 1982]

Appendix A has more details about the releases.

We anticipate Cedar will grow more slowly in the future since most of its planned functionality is in place or under active development. It may grow another 50% in the next year. A major release could then take 50% longer for each phase, although the number of components that change is related to the number of programmer-days between releases and the number of

implementors is expected to remain relatively constant over the next year.

*DF Software*

The need for automated Cedar releases was not anticipated in the early planning for Cedar. BringOver and StoreBack were designed for other reasons. Cedar runs on the Dorado, a high-speed personal computer [Dorado, 1981]. At one time, most users of the Dorado had to share machines without easily removable disk packs (unlike the Xerox Alto). BringOver and StoreBack were developed so programmers could move files from one Dorado to another.

Before the release process was designed, DF files were used to distribute a new version of Cedar. During that period, someone would delete all files from a Dorado and try to find compatible versions of new software from various remote file servers. If successful, he would copy (by hand) all the files to a single directory and (by hand) edit a DF file that described these files. Users could then run BringOver on this DF file. This scheme quickly became unmanageable as the number of components increased. We are certain it would be impossible to do this for Cedar in its current size.

During the four months after the initial Cedar release, the speed of DF software was tuned to take better advantage of protocols to our file servers and to omit unnecessary analysis. The times quoted above for phases one, two, and three leave little room for future improvement, unless we improve the speed of our file servers.

*Release Procedures*

As the size of Cedar releases and the number of programmers responsible for release components has increased, programmers have complained about two design decisions.

1. There is often uncertainty whether a DF file is changing in the next release. Implementors frequently delay sending a message announcing a new component until its changes are completely finished.
2. Implementors who "Import" other DF files are not sure where the working DF files are stored. Although their locations are listed in messages sent announcing new versions, clients often forget to note a change of location. One of the most common errors in DF files submitted to the release is a reference to the wrong version of a DF file.

We have addressed these problems by making some managerial changes and minor changes to StoreBack:

a. We have added a third directory <PreCedar> that is used solely for integration of the next release. We require that all files submitted to the release must be stored on <PreCedar>. All DF files that are changing must be stored on <PreCedar>Top>. All files on the <PreCedar> directory are deleted immediately after they are moved to the <Cedar> directory.
b. We have divided the announcement of a new version of a component into two separate steps for components that are heavily used. Early in the release process the Release Master

determines whether these components will be changing and sends a message listing those that will and those that will not change. The implementors can then announce the final versions as they are ready.

c. Users may take released software, make changes to it, and store their new version on the <PreCedar> directory so it can be released. Those users who want to keep their development versions of files on other directories are free to do so. (They might do that because they want to make incompatible changes to their software after they have submitted it to the release but before the release is finished, or they may not like to copy their software from <Cedar> back to <PreCedar> after all files on <PreCedar> are deleted.) To help them move their files, we provide an option to StoreBack that takes a collection of working DF files and copies them to <PreCedar> as follows: If the working DF file describes working versions using

    Directory *working*
      ReleaseAs [Indigo]<Cedar>BTree>

  then StoreBack will store a version of this DF file on <PreCedar> with entries like

    Directory [Indigo]<PreCedar>BTree>
      ReleaseAs [Indigo]<Cedar>BTree>

  and copy all files in the working directory to the BTree> subdirectory of <PreCedar>.

These changes solve problems (1) and (2) since new versions of DF files can only be stored on <PreCedar>Top>. If a DF file is not there then it is either not changing or is not ready. Since <PreCedar> is erased after each release there cannot be any old versions of DF software on <PreCedar>. The addition of a special option to StoreBack performs the movement of files to <PreCedar> without requiring any editing of DF files by the implementor. The use of <PreCedar> is shown in Figure 6.

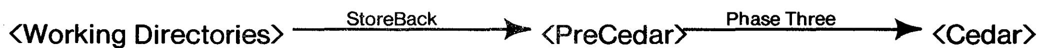



Figure 2.6

<PreCedar> stores *all* files submitted to a release, not just DF files, which increases disk space needs substantially. If only the DF file were stored on <PreCedar>, then programmers who made changes to software stored in their working directories before the release occurred might delete "old" versions that had been changed but were referenced by the DF files on <PreCedar>.

We are also considering changing Phase Three of the Release Tool to *rename* rather than *copy* the files from <PreCedar> to <Cedar>. The Release Tool can copy about 12 megabytes an hour, so complete releases of 70 megabytes can take six hours to copy all the files. Renaming those files would be much faster. We could not consider renaming files as long as the files were stored on private working directories and the user depended on their presence.

The complexity of making releases continues to increase. We recently distributed a memo describing the release process and guidelines for submissions to the release, and no longer fix errors in DF files that are covered by the guidelines, since that was taking more and more time. Peer pressure to conform to the guidelines and the two stages of DF messages (#1 above) should guarantee continued manageability of Cedar releases up to their expected size.

Above a certain level of complexity, however, these solutions would not be sufficient and a more automated integration phase would be necessary. For example, dependency graph and change information in a database could be used to stage the integration phase in a more formal and organized sequence.

*Complete Cedar Bootstraps*

Many aspects of bootstrapping Cedar are simplified when interfaces to the lowest and most heavily used parts of the boot file are not changed. Some major releases use the same versions of interfaces to the Cedar object allocator and fundamental string manipulation primitives. Most major releases use the same versions of interfaces to the underlying Pilot system such as the file system and process machinery. The implementations of these very stable parts of the system may be changed in ways that do not require interface changes.

The two Cedar releases that have included changes to the interfaces of the Pilot operating system forced us to change our style of integration for those releases. Since the released loader cannot load modules that refer to the new versions of Pilot interfaces, the software of Cedar that is pre-loaded in the boot file must all be recompiled before any changes can be tested. Highest priority is given to producing a boot file in which these changes can be tested.

If the DF files describing Cedar were layered in hierarchical order, with Pilot at the bottom, this boot file could be built by producing new versions of the software in each DF file in DF-dependency order. Figure 2.7 (at the end of the chapter) shows the dependency graph for DF files in the boot file, where an arrow from one DF file (e.g., Rigging.DF) to another (e.g., CedarReals.DF) indicates Rigging.DF Imports some file(s) from CedarReals.DF. Some arrows have been omitted in this figure. For example, Rigging.DF also depends on CompatibilityPackage.DF, but the dependency by CedarReals.DF on CompatibilityPackage.DF ensures a new version of Rigging.DF will be made after both lower DF files. The PilotInterfaces.DF file is at the bottom and must be changed before any other DF files.

This dependency graph is not acyclic, however. The most extreme cycle is in the box with six DF files in it, which is expanded in Figure 2.8 (at the end of the chapter). Each DF file is in a cycle with at least one other DF file, so each DF file depends on the other (possibly indirectly) and no DF file can be announced "first". There is an ordering in which these components can be built: If the interfaces listed in each of the DF files are compiled and (partial) DF files containing those interfaces are stored on <PreCedar>, each programmer can then compile the implementation modules in this component and then store the remaining files on <PreCedar>.

The dependency graph for interfaces is shown in Figure 2.9 (at the end of the chapter). This graph indicates that the interfaces of CIFS, VersionMap, Runtime, WorldVM, ListsAndAtoms, and

IO could be compiled in that order. This interface dependency graph had cycles in it in the Cedar 3.3 release that have since been eliminated. Appendix B has copies of some of these DF files before and after the release.

Recompilation of all the interfaces in the boot file (approximately 600 interfaces in Cedar 3.3) requires that at least nine Cedar programmers participate. Since the boot file cannot be produced until all interfaces and implementation modules in the DF files of Figure 2.7 are compiled, we encourage interface changes be made as soon as possible after a successful release and only once per release. Once the programmers have made their interface changes and a boot file using the new interfaces is built, the normal period of testing can occur and new changes to implementation modules can be made relatively painlessly.

Components being released that are outside the boot file have a much simpler dependency structure, shown in Figure 2.10 (at the end of the chapter). The majority of these components are application programs that use Cedar facilities already loaded in the boot file.

*DF Files as a Description Mechanism*

We are beginning to take advantage of the information in the DF files of a release to study and plan the development of the Cedar system. The ability to scan, or query, the interconnection information gives us a complete view of the use of software by other programs in the system. For example, we can mechanically scan the DF files of an entire release and build a dependency graph describing the interfaces used in Cedar and which implementors depend on these interfaces. Since VerifyDF ensures all interfaces needed by a component are described in its DF file, we can be sure we have an accurate database of information. We can use this information to evaluate the magnitude of changes and anticipate which components will be affected. We can also determine which interfaces are no longer used, and plan to eliminate the implementation of those interfaces. This happens often in a system as large as Cedar while it is under active development.

*The Problem of Multiple Modifiers*

The Cedar release/DF approach assumes only one person is changing a DF file at a time. How would we cope with more than one modifier of a package? If the package is easily divided, as with the Cedar window manager and editor, two or more DF files can be Included by an "umbrella" DF file that is released. One of the implementors must "own" the umbrella DF file and must make sure that the versions Included are consistent (by running VerifyDF on the umbrella.) If the package is not easily divided, then either a check in/check out facility must be used on the DF file and its contents to guarantee only one person is making changes at a time, or a merge facility (as in [Rochkind, 1975]) would be needed to incorporate mutually exclusive changes. Should more than one programmer change the same module, this merge facility would have to ask for advice on which of the new versions, if any, to include in the DF file.

*Integration with Version Maps*

The Cedar Runtime system may need to read the symbol table(s) of file(s) when it is asked for details of the structures of Cedar types. The Compiler stamps the reference to these symbol tables with the 48-bit version stamp it computes for each object file. Since the symbol tables are large, they are not normally kept in the released boot file. When a symbol table is needed, the Cedar runtime uses the 48-bit version stamp as an index into a *version map* that gives a file name, including host and directory information. The file containing the symbol table is then retrieved to the local disk. The file name cannot be saved in place of the 48-bit version stamp since, when it is computed, the compiler does not know where the object file containing the symbol table will be stored and whether or not it has been released.

Each boot file automatically retrieves this version map, which is computed by reading the released DF files and computing a map from 48-bit version maps of object files to files stored on the release directory. The construction of this version map is done immediately after all the files are moved to the release directory in phase three of the release. There are two problems with this approach: 1) During the integration phase, the version map must be constructed periodically and distributed to personal machines that are running the pre-release boot file. This information is often out of date. 2) Packages under development and not part of the release do not have entries in the version map.

We intend to solve these problems by changing the way these version maps are constructed:

1. VerifyDF will compute a version map for working versions of software described by DF files. There will be an entry for the version map in each DF file.
2. The Release Tool will read in these version maps and make copies of them in the release directory. These copies will have the file names of the copy of each file in the release directory instead of in the integration directory.
3. When the user runs BringOver on one of the working or released DF files, the version map in the DF file is retrieved and added to a list of version maps to be searched when the Runtime wants to lookup a version stamp.

Associating the version maps with DF files generalizes their use to pre-release files and files that are not in the release.

*Use on other Software Systems*

The algorithms and software described above could be used on other medium- to large-scale software projects. For example, the U.C. Berkeley distribution of UNIX software (4.1 BSD) consists of 450,000 lines of C language code, 2500 files, and takes 10.64 megabytes to store [Joy, 1981]. Another example is the software for the Xerox Star [Harslem-Nelson, 1982], which consists of 255,000 lines of Mesa code, 401 interface, 440 implementation, and 88 configuration modules (these numbers do not include the Pilot operating system.) Compare these numbers to the 425,000 lines and 4500 files of recent Cedar releases.

The Mesa Group of the System Development Division of Xerox is using this software for their internal software development. They have an arrangement of boot files and object files that is similar to that of the Cedar project. Figure 2.11 at the end of this chapter shows the dependency graph for the boot file of their first release. Figure 2.12 is the part of the graph that lies outside their boot file. These figures are analogous to figures 2.7 and 2.10. Their first release consisted of 3490 files, 325,000 lines of Mesa code, and 127 DF files. See Appendix A for statistics on their release.

## 2.8 Conclusions

We have built and demonstrated a system that controls Cedar software in a way that puts a small burden on implementor, user, and Release Master alike, without resorting to specialized skills or dedicating an implementor to the task of organizing and distributing large numbers of files. With approximately five man-months of programming, we have shown that a small amount of automation goes a long way. Doubtless other projects will have other needs and cannot adopt our scheme without modifications. We believe those modifications are not impossible.

In addition to meeting our original goal, that of frequent releases without loss of integrity, we have discovered the benefits of project organization around a machine readable, centrally maintained set of packages.

Releases and DF files are being used to automate one part of the software development cycle in Cedar. DF files are not a replacement for tools to automatically recompile a set of source programs (such as Make [Feldman, 1979]) or other tools to provide fast turnaround for small program changes. Tools for these and other programmer aids are an area for further research. Cedar work in this area is described in Chapters 3 and 4.

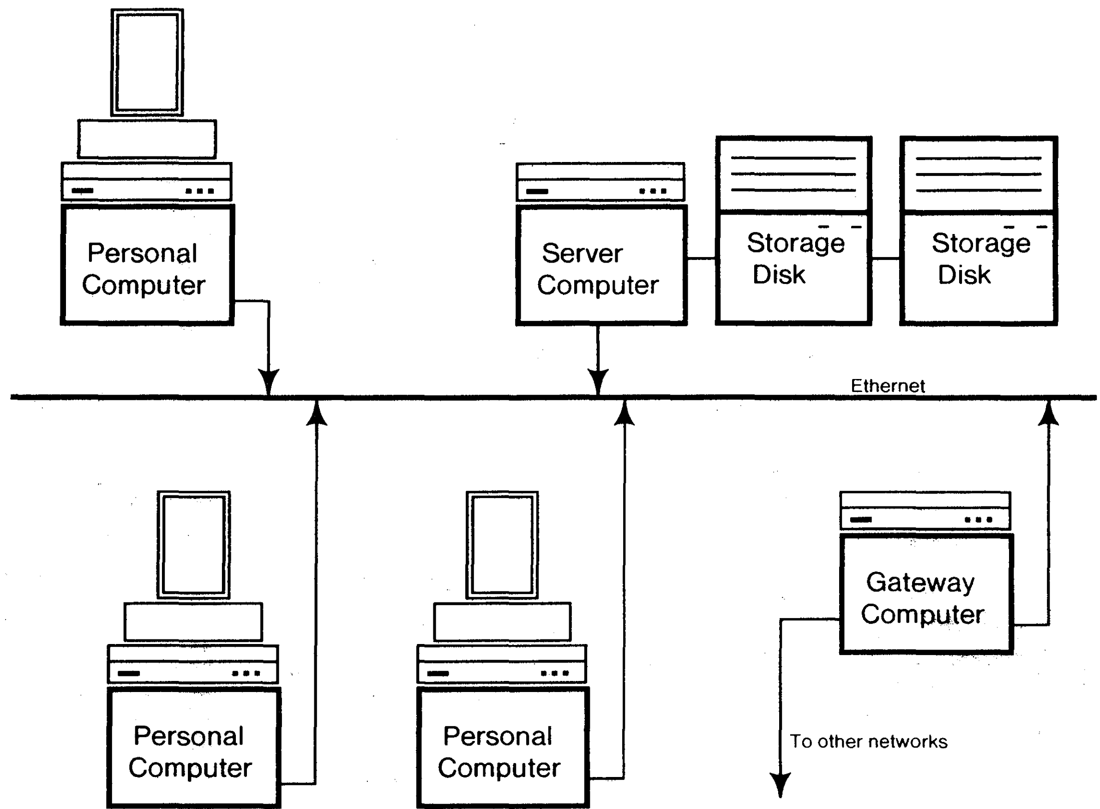
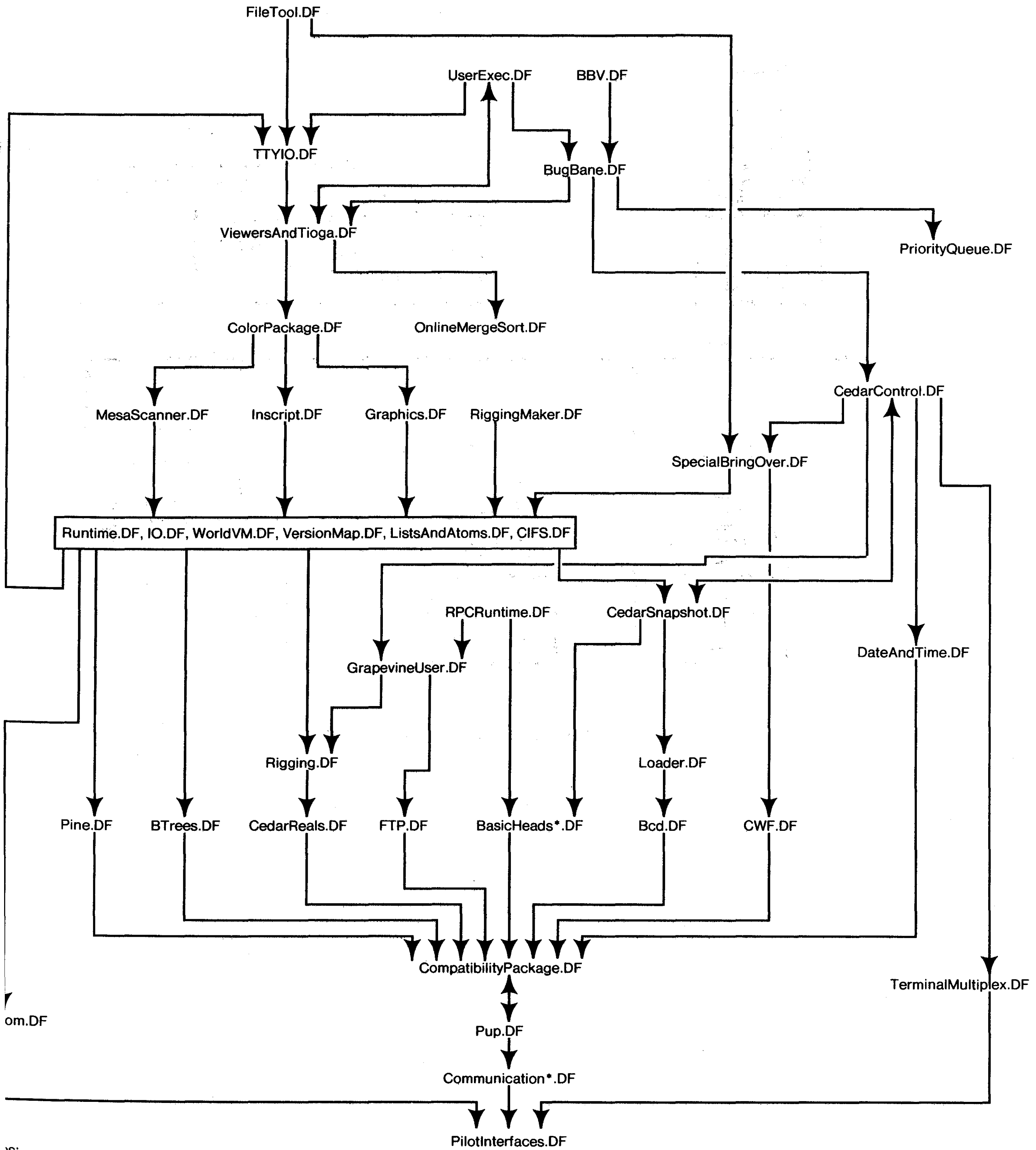



Figure 2.4 Xerox Environment

# Cedar 3.3 DF Files Dependencies (Boot File)

August 29, 1982




ɘs:

Files in tail DF file depend on files in head DF file.

Double-headed arrows indicate mutual dependency.

BasicHeads*.DF stands for BasicHeadsDorado.DF, BasicHeadsD0.DF, BasicHeadsCommon.DF.

Communication*.DF stands for CommunicationPublic.DF, CommunicationFriends.DF, and RS232Interfaces.DF.

`ompatibilityPackage.DF includes MesaBasics.df.

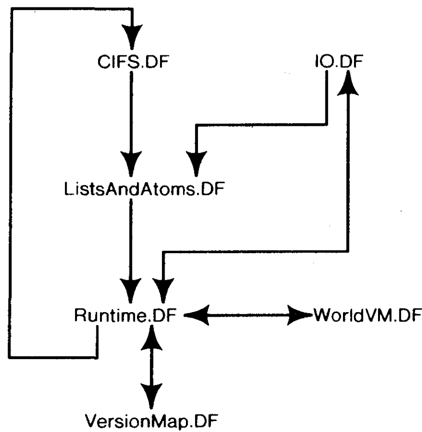
Figure 2.7

# Cedar 3.3 DF Dependencies (Detail in Boot File)

August 29, 1982

| Runtime.DF, IO.DF, WorldVM.DF, VersionMap.DF, ListsAndAtoms.DF, CIFS.DF |
|---|

stands for



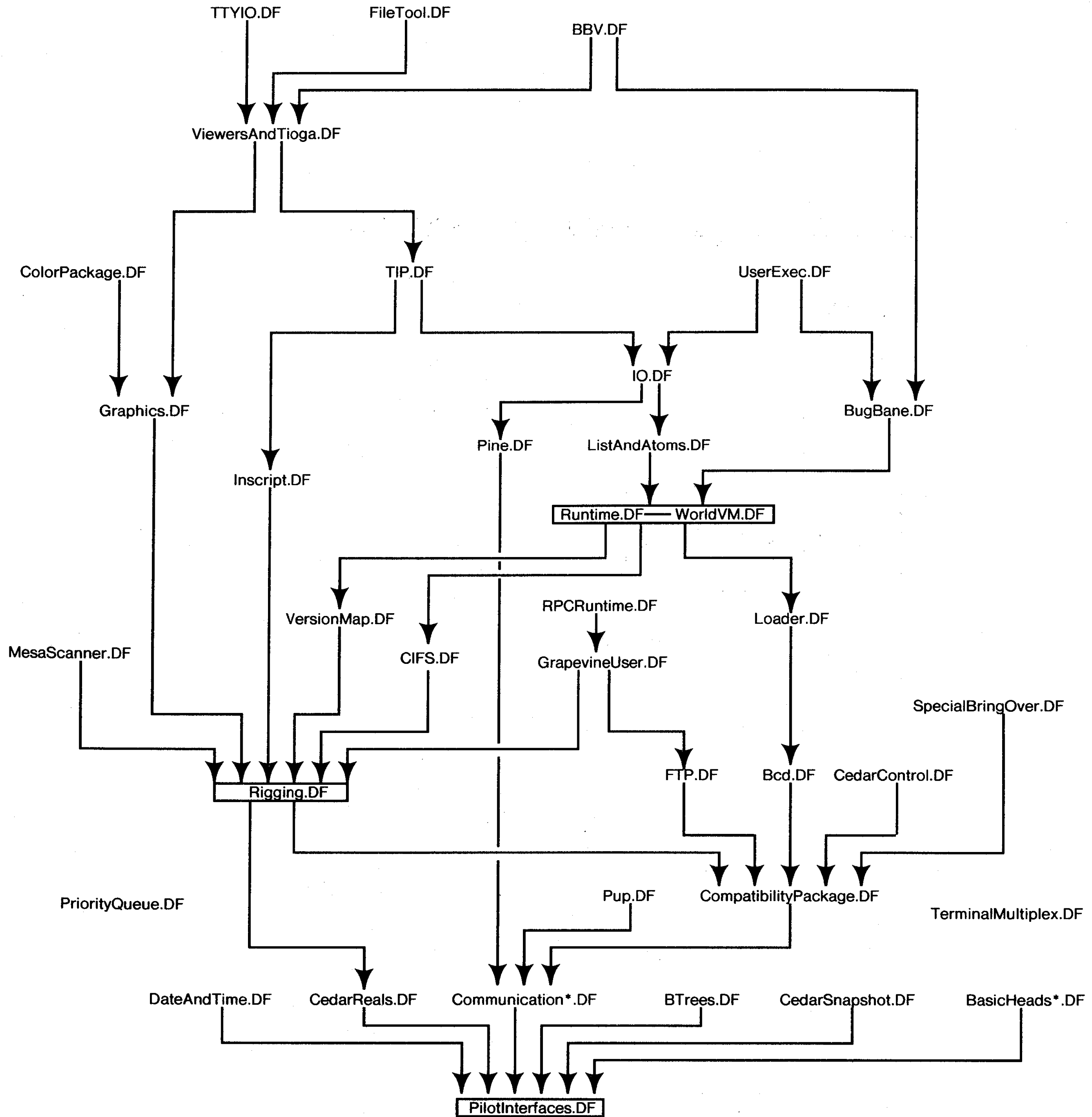

Notes:

Double-headed arrows indicate mutual dependency.

Figure 2.8

# Cedar 3.3 Definitions-DF Dependencies

August 29, 1982




Notes:

Interfaces in tail DF file depend on files in head DF file.

Communication*.DF stands for CommunicationPublic.DF, CommunicationFriends.DF, and RS232CInterfaces.DF.

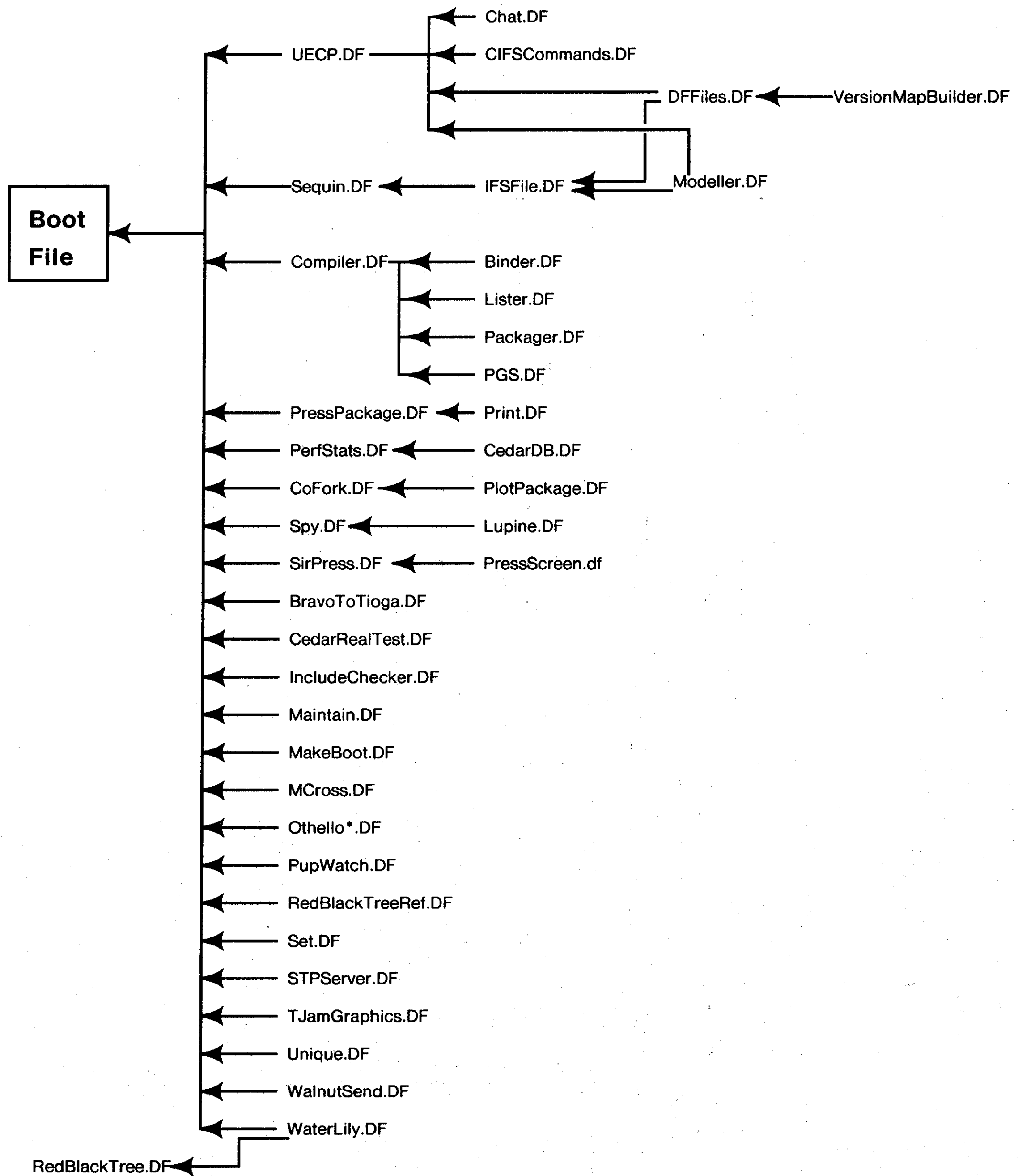
BasicHeads*.DF stands for BasicHeadsDorado.DF, BasicHeadsD0.DF, and BasicHeadsCommon.DF.

CompatibilityPackage.DF includes MesaBasics.DF.

Figure 2.9

# Cedar 3.3 DF Files (Outside Boot File)

August 29, 1982




Notes:

Files in tail DF file depend on files in head DF file.

Othello*.DF stands for OthelloDorado.DF, OthelloD0.DF, and SubOthello.DF.

Figure 2.10

# MG Release 1.0 DF Files Dependencies (Boot File)

September 10, 1982

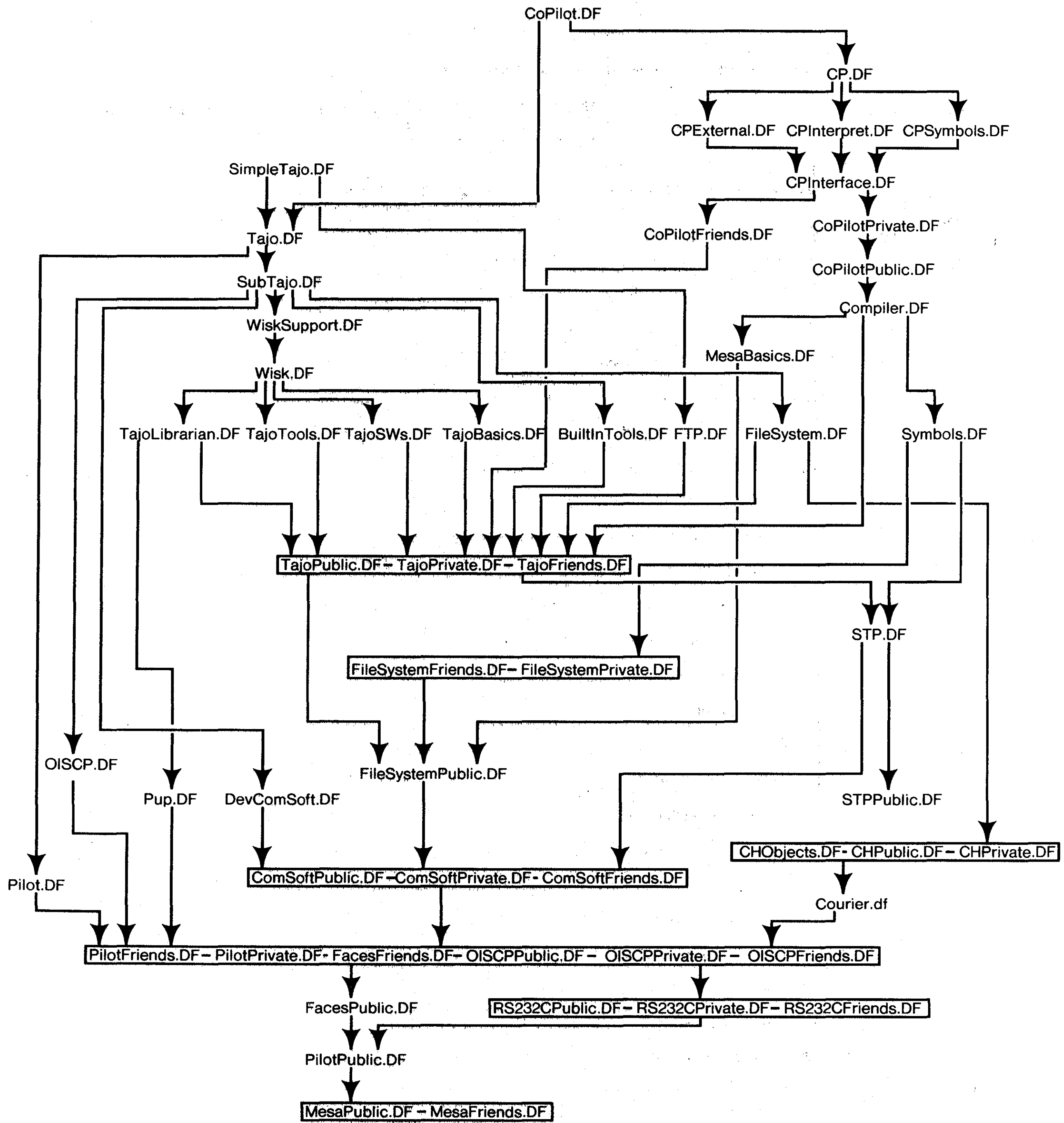
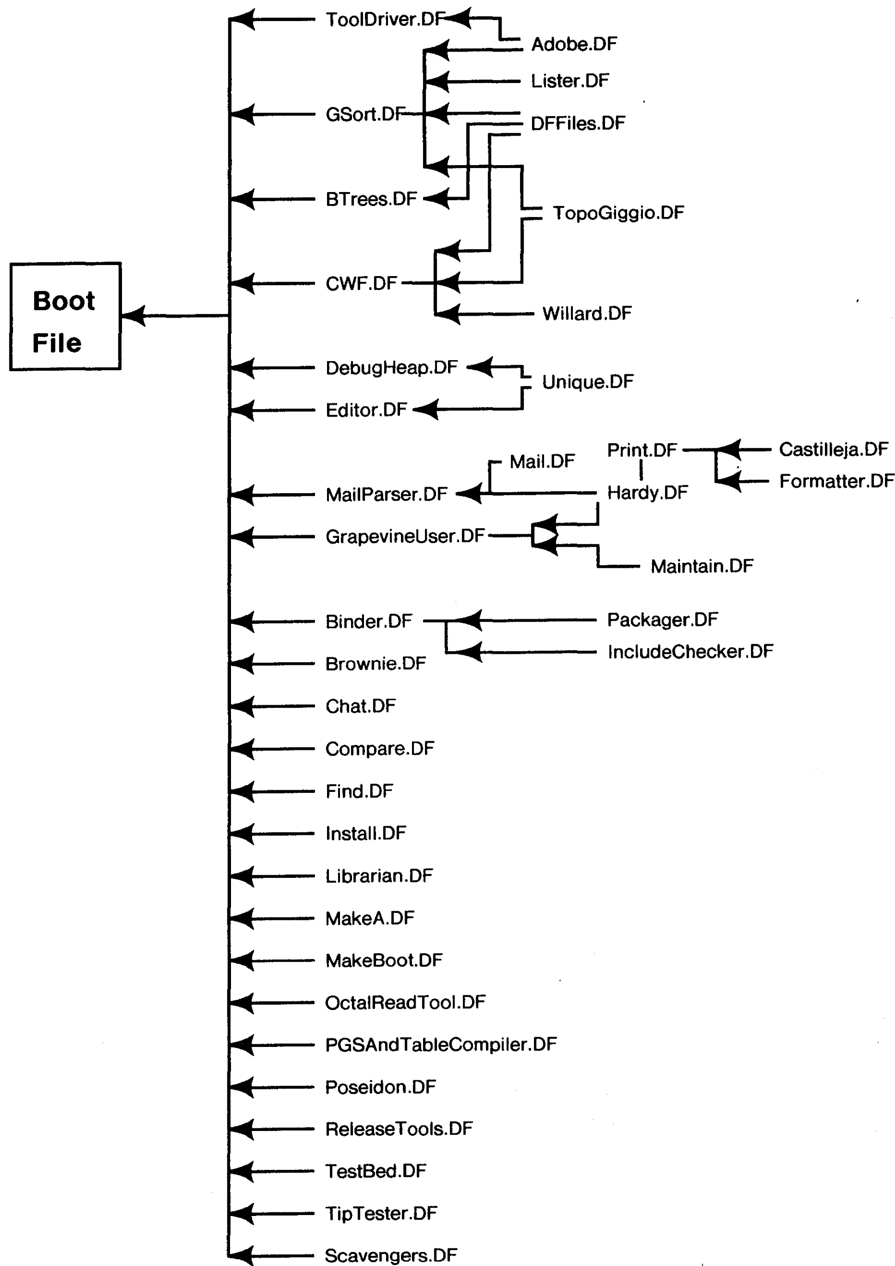



Figure 2.11

# MG Release 1.0 DF Files (Outside Boot File)

September 10, 1982




Notes:

Files in tail DF file depend on files in head DF file.

Figure 2.12

# 3. Practical Use of a Polymorphic Applicative Language

## 3.1 Introduction

This chapter deals with the module interconnection language SML used in a program development system being built as part of the Cedar project [Deutsch-Taft, 1980] of Xerox PARC. SML is a polymorphic and applicative language that is used to describe packages of Cedar modules. The Cedar programmer writes SML programs, which are called *system models*, to specify the modules in his system and the interconnections between them. These system models are analyzed by a program called the *system modeller* that automates the compile-edit-debug cycle by tracking changes to modules and performs the compilation and loading of systems.

We take the view that the software of a system is completely described by a single unit of text. An appropriate analogy is the sort of card deck that was used in the 1950s to boot load and run a bare computer. Note that everything is said explicitly in such a system description: there is no operator intervention (to supply compiler switches or loader options) after the "go" button is pressed. In such a description there is no issue of "compilation order", and "version control" is handled by distributing copies of the deck with a version number written on the top of each copy.

The text of such a system naturally will have internal structure appropriate to the machine on which it runs as well as to the software system itself. Given that in 1982 our system is composed of modules that are stored as text in files, we choose to describe our system in terms of these modules or *objects*. In Cedar, these objects are Cedar modules or Cedar system models. This representation is convenient for users to manipulate; it allows sharing of identical objects, and facilitates the separate compilation of objects. But it is important to appreciate that there is nothing essential in such a representation; in principle, a system can always be expressed as a single text unit.

Whatever representation is chosen for the objects used to describe a system, we require that objects be *immutable*. By this we mean that 1) each object has a unique name (*unique-id*), and 2) the contents of an object never change once the object is created. Objects are immutable but they can be destroyed by deletion.

System models refer to the objects of a system. Since objects are immutable, we use the unique name of the object in place of its contents. Information in databases is indexed by these unique-ids. Use of models with unique-ids for these destroyed objects will give errors, but no confusion will result.

A system model is a stable, unambiguous representation for a system. It is easily transferred among programmers and file systems. It has a readable text representation that can be edited by a user at any time. Finally, it is usable by other program utilities such as cross-reference programs, debuggers, and optimizers that analyze inter-module relationships.

In this chapter we focus on the SML language constructs and what they mean. Chapter 4 describes the use of SML as part of the System Modeller. The SML language and the modeller are presented separately since the modeller may handle (in the future) descriptions of software systems written in other programming languages. Cedar has one of the most complicated and powerful module interconnection systems; the language to describe most other language's module interconnections would be much simpler. We will see that SML can express a very rich set of interconnections. Note that it is not necessary to understand all of Chapter 3 to understand Chapter 4, or vice versa. Readers primarily interested in one chapter need only read the first few sections of the other.

The specification of module interconnection facilities of Cedar requires use of *polymorphism*, where the specification can compute a value that is later used as the type for another value. This kind of polymorphism is explained in detail later. The desire to have a crisp specification of the language and its use of polymorphism led us to base SML on the Cedar Kernel language, which is used to describe the semantics of Cedar programs.

The semantics of the SML language have to be unambiguous so every syntactically-valid system model has clear meaning. The Cedar Kernel language has a small set of principles and is easily implemented. The clear semantics of Kernel language descriptions give a concise specification of the SML language and give good support to the needs of the module interconnection specification. SML could have been designed without reference to the Kernel language. However, without the Kernel language as a base, there would be less confidence that all language forms had clear meaning.

SML is an *applicative* language, since it has no assignment statement. Names (or identifiers) in SML are given values once, when the names are declared, and the value of a name may not be changed later unless the name is declared in some inner scope. SML is easier to implement because it is applicative and function invocation has no side effects.

This chapter describes the existing module interconnection structures of the Cedar language and justifies the need for a new language to describe Cedar systems. The fundamental concepts of SML are presented, followed by a description of SML's treatment of files. The Cedar Kernel language, which serves as a basis for SML, is described, followed by a section on the syntax and semantics of SML expressions. This chapter concludes with notes on implementation, a discussion of problems and experience we have gained, and a section of extended examples.

### 3.2 Existing Module Interconnection Facilities

Cedar is based on the Mesa language [Mitchell, *et al.*. 1979], [Lauer-Satterthwaite, 1979]. Cedar contains features for automatic storage management (garbage collection) and allows binding of types at runtime (pointers to objects whose types are known only at runtime). Cedar inherited from Mesa a rich module interconnection structure that provides information hiding and strong type checking at the module level, rather than at the procedure level. In order to understand the motivation for SML, it is important to know about the existing module interconnection facilities in

Cedar.

A Cedar system consists of a set of modules, each of which is stored in a separate file. A module can be one of two types: an implementation (PROGRAM) module, or an interface (DEFINITIONS) module. Interface modules contain constants found in other Pascal-like languages: procedure declarations, type declarations, and other variables. A module that wishes to call a procedure declared in another module must do so by IMPORTing an interface module that declares this procedure. This interface module must be EXPORTed by a PROGRAM module. For example, a procedure USortList declared in a module SortImpl would also be declared in an interface Sort, and SortImpl would EXPORT Sort. A PROGRAM that wants to call the procedure USortList does so by IMPORTing Sort. We call the importer of Sort the "client" module and say SortImpl (the exporter) "implements" Sort. Of course, SortImpl may IMPORT interfaces to use that are defined elsewhere.

These interconnections are shown in Figure 3.1, which shows filenames for each module in the upper left corner. The interface Sort defines an object composed of a pair of x,y coordinates. The exporter, SortImpl.Mesa, declares a procedure that takes a list of these objects and sorts them, eliminating duplicates. (LIST in Cedar is a built-in type with a structure similar to a Lisp list.) ClientImpl.Mesa defines a procedure that calls USortList to sort a list of such objects. (Details about the CompareProc have been omitted.)

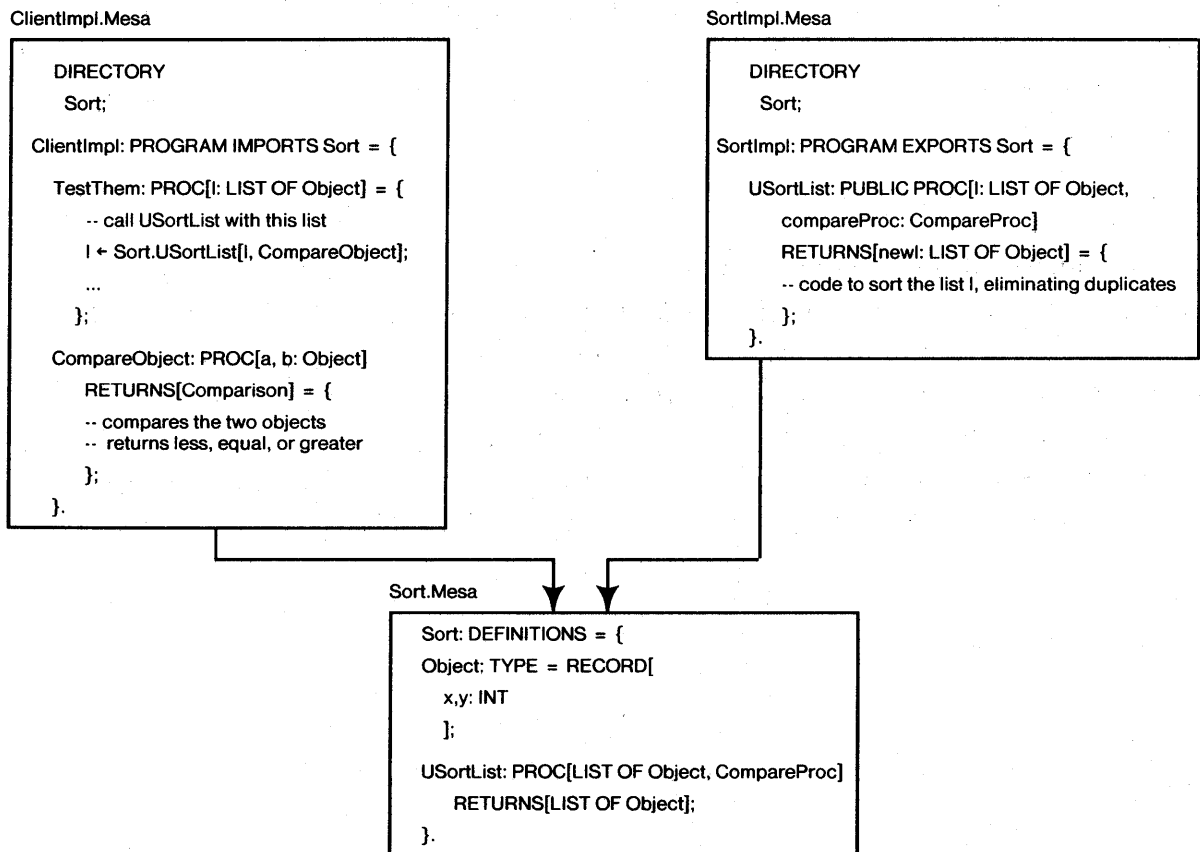



Figure 3.1

*Interface Type Parameterization*

Most collections of modules in Cedar use the same version of interfaces, e.g., there is usually only one version of the interface for the BTree package in a given system. Situations arise when more than one version is used in a system. For example, there could be two versions of an interface to a list manipulation system, each one manipulating a different type of object.

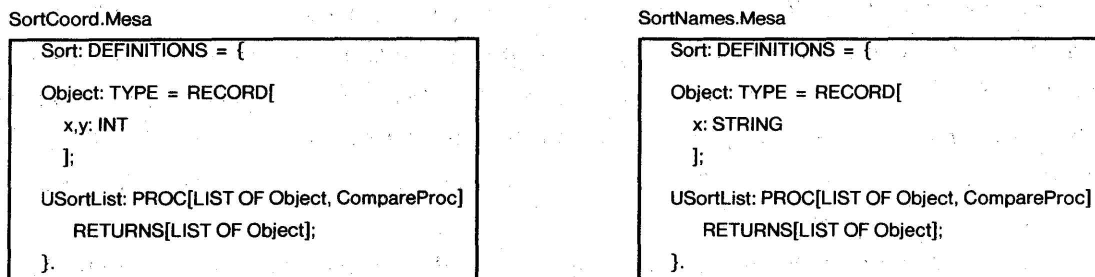



Figure 3.2

Figure 3.2 shows, on the left, the module from Figure 3.1, and, on the right, a similar module that defines an "Object" to be a string instead of coordinates. A module that refers to the Sort interface would have to be compiled with one of the two versions of the Sort interface, since the compiler checks types of the objects being assembled for the sort. We call this *interface type parameterization*, since the types of items from the interface used by a client (ClientImpl.Mesa) are determined by the specific version of the interface (SortCoord.Mesa or SortNames.Mesa).

*Interface Record Parameterization*

A different kind of parameterization may occur when two different implementations for the *same* interface are used. For example, a package that uses the left version of the Sort interface in Figure 3.2 above might use two different versions of the module that EXPORTs Sort, one of which uses the QuickSort algorithm and the other uses the HeapSort algorithm to perform the sort. Such a package includes both implementors of Sort and must specify which sort routine the clients (IMPORTers) use when they call Sort.USortList[]. In Cedar and Mesa it is possible for a client module to IMPORT *both* versions, as shown in Figure 3.3.

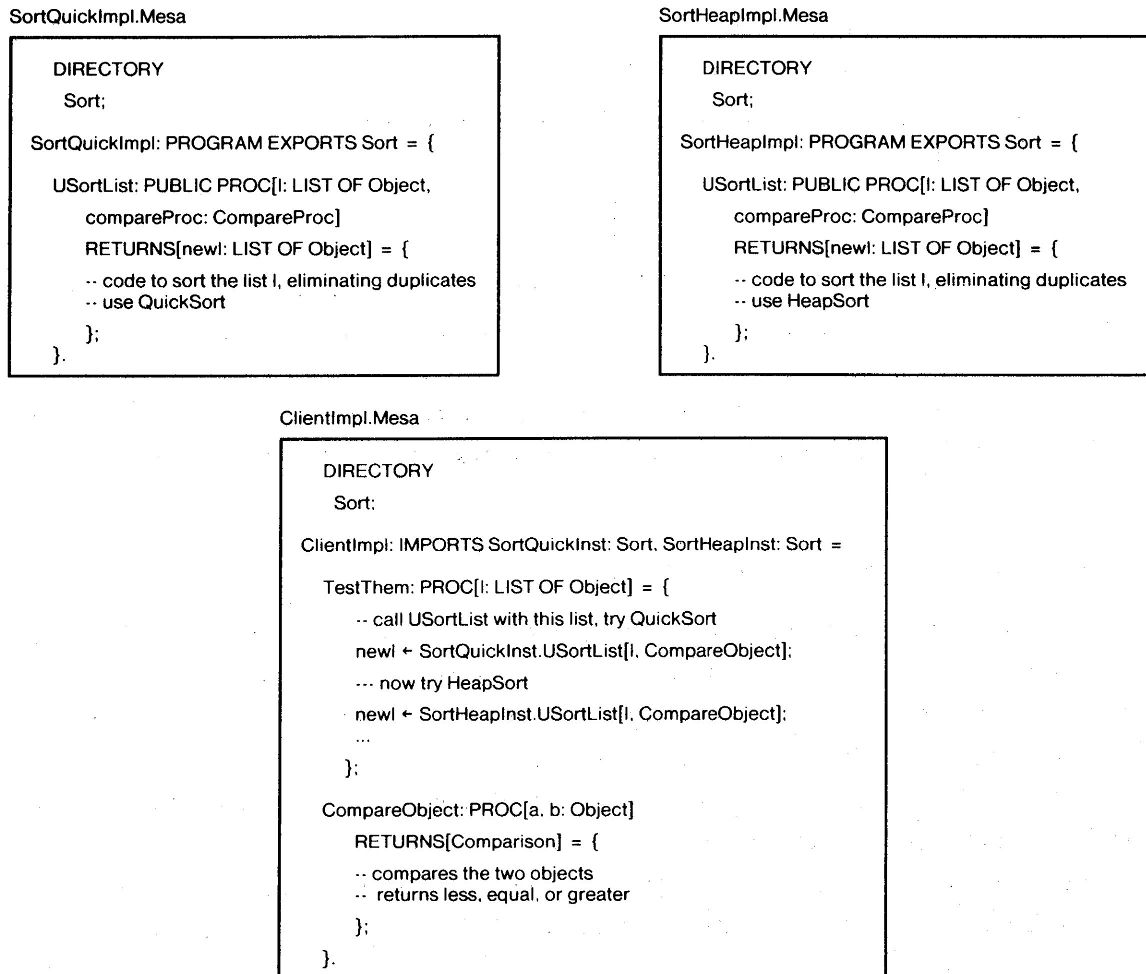



Figure 3.3

In Figure 3.3, SortQuickImpl and SortHeapImpl both EXPORT different procedures for the Sort interface. One procedure (SortQuickImpl) uses QuickSort to sort the list. The other uses HeapSort to sort the list. The importer, ClientImpl, imports each version under a different name (SortQuickInst and SortHeapInst are called *interface records*, since they are represented as records containing pointers to procedures). The client procedure "TestThem" calls each in turn by specifying the name of the interface and the name of the procedure (e.g., SortQuickInst.USortList[]).

One step remains: how are the two interface records that are exported by SortQuickImpl and SortHeapImpl connected to the two interface records (SortQuickInst and SortHeapInst) required by ClientImpl? A program called the Mesa Binder makes these connections by reading a specification written in a subset of Mesa called C/Mesa. C/Mesa source files, called CONFIGURATIONS, name the implementation modules involved and specify the interconnections. Figure 3.4 shows the configuration that makes the connection in our example:

```
ClientConfig: CONFIGURATION = {
    SQI: Sort ← SortQuickImpl[];
    SHI: Sort ← SortHeapImpl[];
    ClientImpl[SortQuickInst: SQI, SortHeapInst: SHI];
    }.
```

Figure 3.4

Two variables are declared (SQI and SHI) that correspond to the interface records exported by the two modules. The client module is named, followed by the two interfaces given in keyword parameter notation.

This is called *interface record parameterization*, since the behavior of the client module is a function of which interfaces SortQuickInst and SortHeapInst refer to when they are called in ClientImpl.

*Why a New Module Interconnection Language?*

C/Mesa, as currently defined, cannot express interface type parameterization at all. The semantics of some C/Mesa specifications are ambiguous. In addition, our program development system uses version management and file location information in system descriptions. We chose to replace the use of C/Mesa in Cedar by the use of SML.

SML programs give the programmer the ability to express both kinds of parameterization. It is possible to think of SML as an extension of C/Mesa, although their underlying principles are very different. Before explaining SML, we give an example of modules that use both interface type and interface record parameterization and show how this can be expressed in SML.

### 3.3 Example of SML Program

The essential features of SML are illustrated by the following simple model and are discussed in the next section on SML's treatment of files. A complete description of the SML language is in section 3.7.

Consider two versions of the Sort interface (from Figure 3.2), and two EXPORTers of Sort (from Figure 3.3). Since the exporters do not depend on the kind of object (coordinates or names), the exporters can each be constructed with a different type of object. Assume the client module wants to call USortList with all four combinations of object type and sort algorithm: (coördinates+quicksort, coordinates+heapsort, names+quicksort, names+heapsort). Figure 3.5 shows a version of ClientImpl that uses all four.

ClientImpl.Mesa

```
DIRECTORY
   SortCoord: INTERFACE Sort,
   SortNames: INTERFACE Sort;

ClientImpl: PROGRAM IMPORTS SortQuickCoordInst: SortCoord,
         SortQuickNamesInst: SortNames, SortHeapCoordInst: SortCoord,
         SortHeapNamesInst: SortNames = {

   TestThem: PROC[l1: LIST OF SortCoord.Object, l2: LIST OF SortNames.Object] = {
         newl ← SortQuickCoordInst.USortList[l1, CompareCoordinateObjects];
         newl ← SortHeapCoordInst.USortList[l1, CompareCoordinateObjects];
         newl ← SortQuickNamesInst.USortList[l2, CompareNameObjects];
         newl ← SortHeapNamesInst.USortList[l2, CompareNameObjects];
         };

   CompareCoordinateObjects: PROC[a, b: SortCoord.Object] RETURNS[Comparison] = {
         -- compares a and b, returns less, equal, or greater
         };

   CompareNameObjects: PROC[a, b: SortNames.Object] RETURNS[Comparison] = {
         -- compares a and b, returns less, equal, or greater
         };

}.
```

Figure 3.5

In SML, a model to express this is shown in Figure 3.6.

```
ClientModel ~ [
   -- interface types
   SortCoord: INTERFACE ~ @SortCoord.Mesa[];
   SortNames: INTERFACE ~ @SortNames.Mesa[];
   -- interface records
   SQCI: SortCoord ~ @SortQuickImpl.Mesa[SortCoord];
   SQNI: SortNames ~ @SortQuickImpl.Mesa[SortNames];
   SHCI: SortCoord ~ @SortHeapImpl.Mesa[SortCoord];
   SHNI: SortNames ~ @SortHeapImpl.Mesa[SortNames];
   -- give all to client
   Client: CONTROL ~ @ClientImpl.Mesa[SortCoord, SortNames, SQCI,
            SQNI, SHCI, SHNI]
   ]
```

Figure 3.6

SML allows names to given types and bound to values. After the header, two names

SortCoord and SortNames are given values that stand for the two versions of the Sort interface. Each has the same type, since both are versions of the Sort interface. (Their type is "INTERFACE Sort", where "INTERFACE" is a reserved word in SML and "Sort" is the interface name.) The next four lines bind four names to interface records that correspond to the different sort implementations. SQCI is a name of type "SortCoord" and has as value the interface record with a procedure that uses QuickSort on objects with coordinates. Similarly, SQNI has as value an interface record with a procedure for QuickSort on objects with strings, etc. Note that each of the four implementations is parameterized by the correct interface, indicating which type to use when the module is compiled.

The last line specifies a name "Client" of reserved type "CONTROL" and gives it as value the source file for ClientImpl, parameterized by *all* the previously defined names. The first two, SortCoord and SortNames, are values to use for the names "SortCoord: INTERFACE Sort" and "SortNames: INTERFACE Sort" in the DIRECTORY clause of ClientImpl. The last four, in order, give interface records for each of the four imports.

There are a number of nearly-equal names in the example. If all related names were uniform (e.g., SortQuickCoordImpl instead of SQHI and SortQuickCoordInst, and SortHeapCoordImpl instead of SQHI and SortHeapCoordInst) then the parameter lists in the example could be omitted.

The evaluator computes the values and binds them in a linear fashion in example 3.6 above. The kinds of values in SML follow naturally from the objects being represented: the value of "@SortCoord.Mesa[]" is the object file for the interface module SortCoord.Mesa when it is compiled. The value of "@SortQuickImpl.Mesa[]" is an interface record produced when the object file for SortQuickImpl.Mesa is loaded. Note there are two versions of the object file for SortQuickImpl.Mesa: one has been compiled with SortCoord as the interface it exports, and the other has been compiled with SortNames as the interface it exports.

It is helpful to differentiate the two types of parameterization by the difference in uses: Interface type parameterization is applied when a module is compiled and the types of the various objects and procedures are checked for equality. Interface record parameterization is applied when a module is loaded and the imports of other modules are resolved. The interface records by which a module is parameterized (either in C/Mesa or SML) are used to satisfy these inter-module references.

This is a contrived example; the need for this amount of interface parameterization has not arisen and, if it had, the existing Cedar Binder cannot process the analogous C/Mesa description. Each of interface type and interface record parameterization occur quite often (separately), however.

### 3.4 Fundamentals of SML

The SML language is built around four concepts:

1. *Application:* The basic method of computing.
2. *Values:* Everything is a value, including types (polymorphism) and functions.
3. *Binding:* Correspondence between names and values is made by binding.
4. *Groups:* Objects can be grouped together.

*Application*

The basic method of computation in the SML language is by applying a function to argument values. A function is a mapping from argument values to result values.

A function is implemented either by a primitive supplied by the language (whose inner workings are not open to inspection) or by a *closure*, which is the value of a λ-expression whose body, in turn, consists of applications of functions to arguments. In SML, λ-expressions have the form

λ [ free-variable-list ] ⇒ [ returns-list ] IN [ body-expression ]

For example, a λ-expression could look like

λ [x: STRING, y: STRING] ⇒ [a: STRING] IN [ *exp* ]

where "x" and "y" are the free variables in the λ-expression, "a" is the name of the value returned when this λ-expression is invoked, and *exp* is any SML expression that computes a value for name "a". "IN" is like "." in standard λ-notation. It is helpful to think of a closure as a program fragment that includes all values necessary for execution except the λ's parameters, hence the term closure. Every λ-expression must return values, since the language has no side effects. Application is denoted in programs by expressions of the form *f[arg, arg, ... ]*.

*Values*

An SML program manipulates values. Anything that can be denoted by a name or expression in the program is a value. Thus strings, functions, interfaces, and types are all values. In the SML language, all values are treated uniformly, in the sense that any can be

passed as an argument,

bound to a name, or

returned as a result.

These operations must work on all values so that application can be used as the basis for computation and λ-expressions as the basis for program structure. In addition, each particular kind or *type* of value has its own primitive functions. Some of these (like equality) are defined for most types. Others (like subscripting) exist only for specific types (like groups). None of these operations, however, is fundamental to the language.

*Groups*

There is a basic mechanism for making a composite value out of several simpler ones. Such a composite value is called a group, and the simpler ones are its *components* or *elements.* Thus [3, x+1, "Hello"] denotes a group, with components 3, x+1, and "Hello". The main use of groups is for passing arguments to functions without naming them (these are sometimes called *positional* arguments; bindings are used to pass named or keyword arguments, see the next section). Groups are similar to other language's "structures" or "records": ordered, typed sequences of values.

*Binding, Scope, and Declaration*

A *binding* is an ordered set of [name, type, value] triples, often denoted by a *constructor* like this: [x: STRING ~ "s", y: STRING ~ "t"], or simply [x ~ "s", y ~ "t"]. Individual components can be selected from a binding using the "." operation, similar to Pascal record selection: *binding.element* yields the value of the component named "*element*" in *binding.*

A *scope* is a region of the program in which the value bound to a name does not change. For each scope there is a binding that determines these values. A new scope is introduced by a [ ... ] constructor for a declaration or binding, or a LET statement (see below).

A *declaration* is an ordered set of [name, type] pairs, often denoted [x: STRING, y: STRING]. A declaration can be instantiated (e.g. on block entry) to produce a binding in which each name is bound to a name of the proper type. If *d* is a declaration, a binding *b* has type *d* if it has the same names, and for each name *n* the value *b.n* has the type *d.n.*

In addition to the scopes defined by nested bindings, a binding can be added to the scope using a LET statement

LET *binding* IN *expr*

that makes the names in *binding* accessible in *expr* without qualification.

*Role of Types*

Every name has a type, either because the name is in a binding or the name is in a declaration. Names are given values using bindings. If a name is given an explicit type in the binding, the resulting value must have that type. For example,

n: t ~ v

the type of "v" must be "t". Similarly, if "p" is a λ-expression with "a" as a free variable of type "STRING", then

p[b]

type-checks if "b" has type "STRING". There are no restrictions on use of types as values in SML. For example,

```
[n1: t ~ v1,
 n2: n1 ~ v2]
```

declares a name "n1" with a type t and a value v1, and then declares a name "n2" with type "n1" and value "v2". Although each such value can in turn be used as the type of another name, the Modeller implementation does not attach semantics to all such combinations.

### 3.5 Treatment of Values in SML

*Strings*

Strings are useful in a module interconnection language for compiler options and as components of file names. SML contains facilities to declare strings. For example, the binding

```
[x: STRING ~ "lit",
 y: STRING ~ x]
```

gives x and y the string literal value "lit".

*Files*

SML describes software by specifying a file containing data. This file is named in SML by a filename preceded by an @. SML defines @ as source-file inclusion: The semantics of an @-expression are identical to those of an SML program that replaced the @ expression by its contents. For example, if the file inner.sm contained

```
"lit"
```

which is a valid SML expression, the binding

```
[x: STRING ~ @inner.sm,
y: STRING ~ x]
```

is identical in value to the previous example. Since SML is applicative, such inclusion is also equivalent to

```
[x: STRING ~ @inner.sm,
y: STRING ~ @inner.sm]
```

and

```
[x: STRING ~ "lit",
y: STRING ~ "lit"]
```

The @ expression is used in SML to refer to source modules. Although we cannot substitute the @-expression by the contents of the source file since it is written in Cedar, we treat the Cedar source file as a value in the language with a type. This type is (almost always) a procedure type.

The values in SML that describe module interconnection are all obtained by invoking one of the procedure values defined by an @-expression.

*Module Information*

When compiling a Cedar module, all interfaces it depends on (or references) must be compiled first and the compiler must be given unambiguous references to those files. In order to load a module, all imports must be satisfied by filling in indirect pointers used by the microcode with references to procedure descriptors exported by other modules. We describe both kinds of information in SML by requiring that the user declare objects corresponding to an interface file (for compilation) or an interface record with procedure descriptors (for loading), and then parameterize module objects in SML as appropriate.

*Compilation Parameterization*

Consider an interface that depends on no other interfaces, i.e., it can be compiled without reference to any files. SML treats the file containing the interface as a function whose closure is stored in the file. The procedure type of this interface is for a procedure that takes no parameters and returns one result, e.g.,

[] → [INTERFACE Sort]

where "Sort" is the name of the interface, as in Figure 3.1. The application of this λ-expression (with no arguments) will result in an object of type "INTERFACE Mod".

Id: INTERFACE Sort ~ @Sort.Mesa[]

declares a variable "Id" that can be used for subsequent dependencies in other files. An interface "BTree" defined in the file "BTree.Mesa" that depends on an interface named "Sort" would have a procedure type like

[INTERFACE Sort] → [INTERFACE BTree]

The parameters and results are normally given the same name as the interface type they are declared with, so the procedure type would be

[Sort: INTERFACE Sort] → [BTree: INTERFACE BTree]

In order to express this in his model, the user would apply the file object to an argument list

Sort: INTERFACE Sort ~ @Sort.Mesa[];
BTree: INTERFACE BTree ~ @BTree.Mesa[Sort];

These interfaces can be used to reflect other compilation dependencies.

*Interface Record Parameterization*

An interface that is EXPORTed is represented as an interface record that contains procedure descriptors, etc. These procedures are declared both in the interface being exported and in the exporting PROGRAM module. We can think of the interface record as an instance of a record declared by the interface module. Consider the implementation module SortImpl.Mesa in Figure 3.1. SortImpl exports an interface record for the Sort interface and calls no procedures in other modules (i.e., has no IMPORTS). This file would have as procedure type

[Sort: INTERFACE Sort] → [SortInst: Sort]

and would be used as follows:

Sort: INTERFACE Sort ~ @Sort.Mesa[];
SortInst: Sort ~ @SortImpl.Mesa[Sort];

which declares an identifier "SortInst" of type "Sort", whose value is the interface record exported by SortImpl.Mesa. If SortImpl.Mesa imported an interface record for "BTree," then the procedure type would be

[Sort: INTERFACE Sort, BTree: INTERFACE BTree, BTreeInst: BTree] → [SortInst: Sort]

and the exported record would be computed by

SortInst: Sort ~ @SortImpl.Mesa[Sort, BTree, BTreeInst];

where [Sort, BTree, BTreeInst] is a group that is matched to parameters of the procedure by position. Keyword matching of actuals to formals can be accomplished through a binding described in section 3.7.

*Scopes and LET Statements*

LET statements are useful for including definitions from other SML files. A set of standard Cedar interfaces could be defined in the file CedarDefs.Model:

```
[
  Rope: INTERFACE Rope ~ @Rope.Mesa,
  IO: INTERFACE IO ~ @IO.Mesa,
  Space: INTERFACE Space ~ @Space.Mesa
]
```

Then a LET statement like

LET @CedarDefs.Model IN [ *expression* ]

is equal to

LET [

```
    Rope: INTERFACE Rope ~ @Rope.Mesa,
    IO: INTERFACE IO ~ @IO.Mesa,
    Space: INTERFACE Space ~ @Space.Mesa
  ] IN [ expression ]
```

and makes the identifiers "Rope", "IO", and "Scope" available within [ *expression* ].

### 3.6 Complete SML Description

*Syntax*

SML is described by the BNF grammar below. Whenever "x, ..." appears, it refers to 0 or more occurrences of x separated by commas. "|" separates different productions for the same non-terminal. Words in which all letters are capitalized are reserved keywords. Words that are all lower case are non-terminals, except for

*id*, which stands for an identifier,

*string*, which stands for a string literal in quotes, and

*filename*, which stands for a string of characters that are legal in a file name, not surrounded by quotes.

Subscripts are used to identify specific non-terminals, so they can be referenced without ambiguity in the accompanying explanation.

```
exp ::= λ [ decl1 ] ⇒ [ decl2 ] IN exp1
    | LET [ binding ] IN exp1
    | exp1 → exp2
    | exp1 [ exp2 ]
    | exp1 . id
    | [ exp, ... ]
    | [ decl ]
    | [ binding ]
    | id
    | string
    | INTERFACE id
    | STRING
    | @ filename
decl ::= id : exp, ...
binding ::= bindelem, ...
bindelem ::= [ decl ] ~ exp1
    | id : exp1 ~ exp2
    | id ~ exp1
```

*Semantics*

A model is evaluated by running a Lisp-style evaluator on it. This evaluator analyzes each construct and reduces it to a minimal form, where all applications of closures to known values have been replaced by the result of the applications (using $\beta$-reduction). The evaluator saves partial values to make subsequent compilation and loading easier. The evaluator returns a single value, which is the value of the model (usually a binding).

The semantics for the productions are:

exp ::= $\lambda$ [ $decl_1$ ] $\Rightarrow$ [ $decl_2$ ] IN $exp_1$

The expression is a value consisting of the parameters and returned names, and the closure consisting of the expression $exp_1$ and the bindings that are accessible statically from exp. The type is "$decl_1 \rightarrow decl_2$". The value of this expression is similar to a procedure variable in conventional languages, which can be given to other procedures that call it within their own contexts. The closure is included with the value of this expression so that, when the $\lambda$-expression is invoked, the body ($exp_1$) will be evaluated in the correct environment or context.

exp ::= LET [ binding ] IN $exp_1$

The current environment of $exp_1$ is modified by adding the names in the binding to the scope of $exp_1$. The type and value of this expression are the type and value of $exp_1$.

exp ::= $exp_1 \rightarrow exp_2$

The value of exp is a function type that takes values of type $exp_1$ and returns values of type $exp_2$.

exp ::= $exp_1$ [ $exp_2$ ]

The value of $exp_1$, which must be a closure, is applied to the argument list $exp_2$ as follows. A binding is made for the values of the free variables in the $\lambda$-expression. If $exp_2$ is a group, then the components of the group are matched by type to the formals of the $\lambda$-expression. (The group's components must have unique types for this option.) If $exp_2$ is a binding then the parameters are given values using the normal binding rules to bind f ~ $exp_2$ where $exp_2$ is a binding and f is the *decl* of the $\lambda$-expression.

There are two cases to consider:

1. The $\lambda$-expression has a closure composed of SML expressions. This is treated like a nested function. The evaluation is done by substitution or $\beta$-reduction: All occurrences of the parameters are replaced by their values. The resulting closure is then evaluated to produce a result binding. The $\lambda$-expression returns clause is used to form a binding on only those values listed in the $\lambda$-expression returns list, and that binding is the value of the function call.

2. If the function being applied is a Cedar source or object file, the evaluator constructs interface types or interface records that correspond to the interface module or to the implementation module's exported interfaces, as appropriate. After the function is evaluated, the evaluator constructs a binding between the returned types in its procedure type and the values of the function call.

exp ::= $exp_1$ . *id*

The $exp_1$ is evaluated and must be a binding. The component with name "*id*" is extracted and its value returned. This is ordinary Pascal record element selection.

exp ::= [ exp, ... ]

A group of the values of the component exp's is made and returned as a value.

exp ::= [ decl ]
decl ::= *id* : exp, ...

Adds names "*id*" to the current scope with type equal to value of exp. A list of decls is a fundamental object.

exp ::= [ binding ]
binding ::= bindelem, ...
bindelem ::= [ decl ] ~ $exp_1$
| *id* : $exp_1$ ~ $exp_2$
| *id* ~ $exp_1$

A bindelem binds the names in decl to the value of $exp_1$. If an *id* is given instead of a decl, the type of *id* is inferred from that of $exp_1$. The binding between the names in decl and the values in $exp_1$ follows the same rules as those for binding arguments to parameters of functions.

exp ::= *id*

*id* stands for an identifier in some binding (i.e., in an enclosing scope). The value of *id* is its current binding.

exp ::= *string*

A string literal like "abc" is a fundamental value in the language.

exp ::= INTERFACE *id*

This fundamental type can be used as the type of any module with module name *id*. Note *id* is used as a literal, not an identifier, and its current binding is irrelevant. The value of this expression is the atom that represents "INTERFACE *id*".

exp ::= STRING

A fundamental type in the language. The value of "STRING" is the atom that represents string types.

exp ::= @ *filename*

This expression denotes an object whose value is stored in file *filename.* If the file is another model, then the string *@filename* can be replaced by the contents of the file. If it is another file, such as a source or object file, it stands for a fundamental object for which the evaluator must be able to compute a procedure type.

### 3.7 Parameters, Defaults, and Projections

Function calls in SML are made by applying a closure to 1) a group or 2) a binding. If the argument is a group, the parameters of the closure are matched to the components by type, which must be unique. If the argument is a binding, the parameters of the closure are matched by name with the free variables. For example, if p is bound to

```
p ~ λ[x: STRING, y: INTERFACE Y] ⇒ [Z: INTERFACE Z] IN [...]
```

then p takes two parameters, which may be specified as a group

```
[
defs: INTERFACE Y ~ @Defs.Mesa[],
z: INTERFACE Z ~ p["lit", Defs]
]
```

where the arguments are matched by type to the parameters of the closure. (The order of "lit" and Defs in the example above does not matter.) The function may also be called with a binding as follows:

```
[
defs: INTERFACE Y ~ @Defs.Mesa[],
z: INTERFACE Z ~ p[x ~ "lit", y ~ Defs]
]
```

which corresponds to keyword notation in other languages. (The order of x and y in the call of p do not matter in this example.)

Since the parameter lists for Cedar modules are quite long, the SML language includes defaulting rules that allow the programmer to omit many parameters. When a parameter list (either a group or a binding) has too few elements, the given parameters are matched to the formal parameters and any formals not matched are given default values. The value for each defaulted formal parameter is the value of a variable defined in some scope enclosing the call with the same name and type as the formal. Therefore, the binding for Z in

```
[
x: STRING ~ "lit",
y: INTERFACE Y ~ @Defs.Mesa[],
z: INTERFACE Z ~ p[]
]
```

is equivalent to "p[x, y]" by the equal-name defaulting rule.

SML also allows *projections* of closures into new closures with parameters. For example,

```
[
Y: INTERFACE Y ~ @Defs.Mesa[],
p1: [Y: INTERFACE Y] → [Z: INTERFACE Z] ~ p["lit"],
Z: INTERFACE Z ~ p1[Y]
]
```

sets Z to the same value as before but does it in one extra step by creating a procedure value with one fewer free variable, and then applies the procedure value to a value for the remaining free variable. The defaulting rules allow parameters to be omitted when mixed with projections:

```
[
X: STRING ~ "lit",
Y: INTERFACE Y ~ @Defs.Mesa[],
p1: [Y: INTERFACE Y] → [Z: INTERFACE Z] ~ p[],
Z: INTERFACE Z ~ p1[]
]
```

Enough parameters are defaulted to produce a value with the same type as the target type of the binding (the type on the left side of the "~"). When the type on the left side is omitted, the semantics of SML guarantee that all parameters are defaulted in order to produce result values rather than a projection. Thus

```
Z ~ p1[]
```

in the preceding examples declares a value Z of type INTERFACE Z and not a projection whose value is a λ-expression.

These rules are stated more concisely below:

If the number of components is less than those required to evaluate the function body a coercion is applied to produce either 1) the complete argument list, so the function body may be evaluated, or 2) a *projection* of the original λ-expression into a new λ-expression with fewer free variables. If the type of the result of "$exp_1[exp_2]$" is supplied, one of 1) or 2) will be performed. When the target type is not given, e.g.,

```
x ~ proc[Y]
```

case 1) is assumed and all parameters of proc are assumed defaulted. For example, the expression

proc: [Y: STRING, Z: STRING] → [r: R],
x: T ~ proc[Y]

binds the result of applying proc to Y to x of type T. If T is a simple type (e.g., "STRING"), then the proc[Y] expression is coerced into proc[Y, Z], where Z is the name of the omitted formal in the $\lambda$-expression and R must equal T. If Z is undefined (has no binding) an error has occurred and the result of the expression is undefined. If T is a function type (e.g., [Z: STRING] → [r: R]), then a new closure is replaced by the value of Y. This closure may be subsequently applied to a value of Z and the result value can be computed. The type of Z must agree with the parameters of the target function type.

### 3.8 Pragmatics

Some extensions to SML have been made to accommodate various "special" uses of Cedar module interconnection facilities.

*Program Transformation Programs*

Cedar programmers may use a number of programs that analyze a source program and produce new source programs. For example, an LALR(1) parser generator preprocessor takes a grammar as input and produces a) a source file that must be compiled and b) a Cedar object file with parsing data, that must be subsequently loaded. Another example is a remote procedure call stub generator [Nelson, 1981] that takes the source for a Cedar interface and produces four source files that must all be compiled. In each of these cases the output files depend on the input file, and if the input file were modified, the preprocessor would have to be run again. Preprocessor dependencies like this are expressed in the modelling language as built-in functions that take objects as arguments and return objects as results. These objects correspond to full file names with version stamps. For example, use of PGS (the parser generator system) would look like

[NewSource: (X:INTERFACE → XImpl: X), NewParseBcd: ([] → NewParseBcd:INTERFACE)] ~
PGS[@[Indigo]<Cedar>Grammar.Mesa!H]

defines a name NewSource that can be used to refer to the name of the new source file and defines NewParseBcd as the name of the file containing parsing data. NewSource names an object that takes an interface type and produces an interface record. NewSource corresponds to a new Cedar source file produced by PGS. NewParseBcd takes no parameters and returns an interface type. This corresponds to the module containing parse tables produced by PGS.

These files are subsequently referred to by using the objects, e.g.,

NewSource[X, Y]

as an expression in the model.

*Compiler Options*

Certain aspects of the Cedar compiler's execution can be controlled by specification of compiler options. These are normally given as command line "switches" or "flags" consisting of a single letter. For example, "j" instructs the compiler to perform a cross-jumping optimization on the code it generates, "b" instructs it to check for array indices that are out of range (called bounds faults), and "n" instructs it to check for dereferencing of pointers that are NIL-valued. Since the behavior of a system depends in part on these options, they are treated as any other parameter. Each Cedar source file has a procedure type, as described in Section 3.5. These include a STRING parameter that can be specified as in

SortQuickCoordImpl: SortCoord ~ @SortQuickImpl.Mesa["j", SortCoord]

If the strings are not given, there is a built-in default of "" for this parameter. The options string is used to modify the compiler's default options.

*Multiple Exports*

We have described systems where there is one exporter of an interface and (possibly more than) one importer. It is possible to have more than one exporter of the same version of an interface and merge the interface records together. This often arises when the single exporter module becomes very large and is split by the programmer, with some procedures exported to the interface by one module and the other procedures defined in the other exporting module. Two operators on interface record values are available: PLUS and THEN (which are from C/Mesa). A composite interface record can be produced by such an operator:

BTreeImplA: BTree ~ @BTreeImplA.Mesa[],
BTreeImplB: BTree ~ @BTreeImplB.Mesa[],
BTreeImpl: BTree ~ BTreeImplA PLUS BTreeImplB

The PLUS operator produces a new value with both procedure pointers from BTreeImplA and from BTreeImplB. It is an error if the same procedure is defined in both interface records. If PLUS is replaced by THEN, then duplicate exports of a procedure are allowed and, in that case, a procedure pointer defined in the left operand is used instead of the procedure pointer defined in the right operand.

*Starting Modules*

After the modules described in a model are loaded, and all inter-module references are resolved, a process begins execution of the new modules. This is called *starting* the module. The program may specify which modules need to be started by adding the binding of a name with type "CONTROL" to each module's binding in the model. For example,

[Abc: CONTROL, SortImpl: Sort] ~ @SortImpl.Mesa[]

defines a name "Abc" (which does not matter, except that it must not conflict with other names), of built-in type CONTROL. Modules that export no interfaces are always bound to such names.

### 3.9 Implementation Comments

The SML evaluator is embedded in a program management system that separates the functions of file retrieval, compilation, and loading of modules. Each of these functions is implemented by analyzing the partial values of the evaluated SML expression. For example, the application of a file to arguments is analyzed to see whether compilation or loading is required. For each of these phases, the evaluator could be invoked on the initial SML expression, but this would be inefficient. Since the SML language has no iteration constructs and no recursively-defined functions, the evaluator can substitute indirect references to SML expressions through @-expressions by the file's contents and can expand each function by its defining expression with formals replaced by actuals.

This process of *substitution* must be applied recursively, as the expansion of a λ-expression may involve expansion of inner λ-expressions. The evaluator does this expansion by copying the body of the λ-expression, and then evaluating it using the scope in which the λ-expression was defined after adding the actual parameters (as a binding) for the function to the scope.

The scope is maintained as a tree of bindings in which each level corresponds to a level of binding, a binding added by a LET statement, or a binding for parameters to a λ-expression.

Bindings are represented as lists of triples of (name, type, value). A closure is represented as a quadruple (list of formals, list of returns, body of function, scope pointer) where the scope pointer is used to establish the naming environment for variables inside the body that are not formal parameters. The @-expression is represented by an object that contains a pointer to the disk file named. A variable declared as INTERFACE *mod* (i.e., an interface type variable), is represented as a (module name, pointer to module file) pair, and a variable given as type and interface type variable (i.e., an interface record variable) is represented as a (pointer to procedure descriptors, pointer to loaded module).

The *substitution property* of Russell [Demers-Donahue, 1980] guarantees that variable-free expressions can be replaced by their values without altering the semantics of Russell programs. Since SML programs have no variables and allow no recursion, the substitution property holds for SML programs as well. This implies that the type-equivalence algorithm for SML programs always terminates, since the value of each type can always be determined statically.

### 3.10 Experience

The SML language, in a slightly different form, has been in use by about five programmers for the past year. The implementation of the language, as then specified, uncovered a number of fundamental problems with the abstract machine on which SML was based. Work on "de-sugaring" the Cedar language has refined the abstract machine specification into the Kernel language over the last three months. The evaluator for the language is being re-written to take advantage of the new SML fundamentals. The largest improvement has been in the treatment of declarations and bindings. Bindings are now separated from declarations and are treated like any other value.

### 3.11 Relationship to Kernel Language

In 1979, seven years after Mesa development started, members of the Cedar project began work on a formal technique for specifying the semantics of Cedar statements. This technique relied upon the notion of an abstract machine that, for every valid statement and expression in Cedar, given an existing environment of variables and their values, would describe the state of this abstract machine in terms of new variables and values after the execution of that statement or expression. The abstract machine was an attempt at discovering *denotational semantics* for Cedar that could be used, for example, to prove that a Cedar program would not cause the garbage collector to break when the program was run.

The development of the abstract machine has produced 1) a Kernel language that is small, precisely defined and intuitively simple, and 2) a set of rules for "de-sugaring" the existing Cedar syntax into an equivalent set of Cedar Kernel statements. The Kernel language is polymorphic and non-applicative. The section "Fundamentals on SML" describes the applicative subset of the Kernel language. The SML language is composed of this applicative subset and objects that can be used to describe Cedar systems. The eventual goal of the Kernel language effort is to change existing Cedar (or to replace Cedar) so that programmers will write their programs in a language based on the Kernel language. This language will have objects appropriate to Cedar programming (such as numbers and arrays) and syntactic "sugar" for built-in functions (such as IF-THEN and FOR). After this is accomplished, Cedar programmers will write programs and describe systems in the same underlying language.

### 3.12 Extended Example

*Example 1*

The B-tree package consists of an implementation module in the file "BTreeImpl.Mesa" and an interface "BTree.Mesa" that BTreeImpl exports. (There is no client of BTree, so this model returns a value for the interface type and record for BTree. Some other model contains a

reference to this model and a client for that interface.) The BTree interface uses some constants found in "Ascii.Mesa", which contains names for the ASCII character set. The BTreeImpl module depends on the BTree interface (since it exports it) and makes use of three standard Cedar interfaces. "Rope" defines procedures to operate on immutable, garbage collected strings. "IO" is an interface that defines procedures to read and write formatted data to a stream, often the user's terminal. "Space" defines procedures to allocate Cedar virtual memory for large objects, in this case the B-tree pages.

```
-- Ex1.Model
LET [
    Rope: INTERFACE Rope ~ @Rope.Bcd,
    IO: INTERFACE IO ~ @IO.Bcd,
    Space: INTERFACE Space ~ @Space.Bcd,
    ] IN
  BTreeProc ~
    λ [RopeInst: Rope, IOInst: IO, SpaceInst: Space]
      ⇒ [BTree: INTERFACE BTree, BTreeInst: BTree]
        IN [
          Ascii: INTERFACE Ascii ~ @Ascii.Mesa,
          BTree: INTERFACE BTree ~ @BTree[Ascii],
          BTreeInst: BTree ~ @BTreeImpl.Mesa[BTree, Rope, IO, Space, RopeInst,
            IOInst, SpaceInst]
          ]
```

This (simple) model stored in the file "Ex1.Model" describes a BTree system composed of an interface "BTree" and an implementation for it. The first three lines declare three names used later. Since they are given values that are object (.bcd) files, they take no parameters. This model assumes those files have already been compiled. Note they could appear as

```
Rope ~ @Rope.Bcd,
IO ~ @IO.Bcd,
Space ~ @Space.Bcd
```

since the types of the three identifiers can be determined from their values. The seventh line binds an identifier "BTreeProc" to a λ-expression with three interface records as parameters. If those are supplied, the function will return 1) an interface type for the BTree system, and 2) an interface record that has that type. Within the body of the λ-expression (its closure) there are bindings for the identifiers "Ascii", "BTree", and "BTreeInst". In all cases, the type could be omitted as well.

The file "Ex1.Model" can be evaluated. Its value will be a binding of BTreeProc to a procedure value. The value is a λ-expression that must be applied to an argument list to yield its return values. Another model might refer to the BTree package by

```
[BTree, BTreeInst] ~ (@Ex1.Model).BTreeProc[RopeInst, IOInst, SpaceInst]
```

*Example 2*

```
-- CedarDefs.Model
   [
      Rope: INTERFACE Rope ~ @Rope.Bcd,
      IO: INTERFACE IO ~ @IO.Bcd,
      Space: INTERFACE Space ~ @Space.Bcd
      ]


-- BTree.Model
LET @CedarDefs.Model IN [
  BTreeProc ~
    λ [RopeInst: Rope, IOInst: IO, SpaceInst: Space]
      ⇒ [BTree: INTERFACE BTree, BTreeInst: BTree]
        IN [
          Ascii: INTERFACE Ascii ~ @Ascii.Mesa,
          BTree: INTERFACE BTree ~ @BTree[Ascii],
          BTreeInst: BTree ~ @BTreeImpl.Mesa[BTree, Rope, IO, Space, RopeInst,
            IOInst, SpaceInst]
          ]
    ]
```

The prefix part is split into a separate file. The BTree.Model file contains 1) a binding that gives a name to the binding in CedarDefs.Model, and 2) a LET statement that makes the values in CedarDefs.Model accessible in the λ-expression of BTree.Model. Dividing Example 1 into two models like this allows us to establish standard naming environments, such as a model that names the commonly-used Cedar interfaces. Programmers are free to redefine these names with their models if they want to. Appendices D and E have examples of models that use models that define standard Cedar and Pilot interfaces.

### 3.13 Summary

SML is used to describe a module interconnection scheme in which polymorphism occurs naturally. SML consists of the applicative subset of the Cedar Kernel language and values that correspond to interfaces, implementations, and other objects manipulated in the Cedar module structure. The union of the Kernel language and the module interconnection language used by Cedar programs is the first step toward the ultimate goal of making the Cedar language polymorphic.

The System Modelling language extends the existing facilities of the C/Mesa configuration language by recognizing a form of polymorphism in Cedar and providing for it in the language. The System Modelling view is that each interface defines a single INTERFACE in the modelling language. The interconnections between modules are expressed in the modelling language by giving each interface record a type that depends on the interface they implement. We can

describe the fundamental interconnections between modules in terms of interface types and interface records of those types.

The language is Algol-like with its normal name scoping and definition. The most common unit of value is a Cedar module, expressed as a filename followed by a numerical unique-id, and then a list of parameters to the module. The filename is used as a hint since the unique-id identifies the file.

The parameters to a module include the interface types and interface records involved in the compilation and execution of the modules, and also whatever other parameters the module may need, such as character strings to specify the compiler options.

An object file is a source file that has been compiled with interface types filled in. Thus, we need recompile a module only when one of its interface types changes.

# 4. SML Language Implementation: A Program Management Tool

## 4.1 Introduction

This chapter describes a program management system, called the *System Modeller*, that automates the edit-compile-debug cycle of programmers. The System Modeller is driven from system models written in the SML language. Chapter 3 described the SML language in terms of the facilities it offers to describe Cedar software. This chapter describes the tool that operates on these models.

An instance of the System Modeller (there can be more than one) running on the programmer's machine maintains a system model and tracks changes the programmer makes to his software. As he makes changes, the modeller re-compiles and re-loads the new version of the system. Notification of his changes is carried out automatically through a connection between the Cedar editor and all instances of Modellers.

System models will also be used in the Cedar release process, as described in Chapter 2. As a result, system models will become all-encompassing descriptions of Cedar systems, used by the individual programmer and also used to describe the Cedar system itself. To make this work in real-time, several databases are maintained as tables with information used to speed analysis of models in the system. These tables are also used and transformed by the release process when applied to models.

Chapter 2 presented a system for version management in the Cedar project. The System Modeller provides the facilities described in Chapter 2 in a more general context. System models have more information than DF files about the software they describe. As a result, the System Modeller can manage the files of a system *as they are changing*, by providing a user interface that is used by the programmer to edit, compile, load and debug his changes interactively. The price of this extra functionality is increased complexity of description of Cedar systems, as seen in Chapter 3, and increased complexity in algorithms to cache information about the system being worked on.

This chapter will first justify the approach taken with system models, especially the requirement that system models refer to files that are *immutable*. We then present the user interface using an example, showing how a change to a module is fielded by the modeller and the system is re-compiled and loaded. We then present the algorithm to maintain the tables used by the modeller, and then the release process as applied to models. This chapter concludes with a discussion of extensions of this approach.

## 4.2 Treatment Of Objects

System models refer to other models or files. The approach taken here depends very strongly on the notion of a model or Cedar module as an *immutable object* that is referenced by system models. Chapter 3 gave motivation for our emphasis on objects. The Cedar system modeller treats files as objects that are considered *immutable.* If a new version of a source file is created, its creation time changes. Models use the creation times to identify specific versions of objects. Once the creation time changes, the modeller views the new version of the source file as a different object.

*Role Of Objects*

When invoked, the modeller uses the objects in a model to determine which modules need to be recompiled. The modeller will get any files it needs and try to put the system together. Since it has unique-ids for all the needed sources, it can check to see if they are nearby. If not, it can take the path name in the model as a hint and, if the file is there, it can be retrieved. The modeller may have difficulty retrieving files, but it will never make a mistake and retrieve the wrong version. Having retrieved as many files as possible, it will compile any source files if necessary, load the resulting binary files, and run the program.

A model normally refers to source files rather than the less-flexible binary files (or object files) produced by the compiler, whose interface types are already bound. The system modeller takes the view that these binary files are just *accelerators,* since every binary file can be compiled using the right source files and parameters. The model has no entry for a binary file when the source file it was compiled from is listed. Such an entry is unnecessary since the binary file can always be reconstructed from the source. Of course, wholesale recompilation is time-consuming so various databases are used to avoid unnecessary recompilation.

The Modeller takes a very conservative approach, so the users and distributors can be sure there is no confusion over which versions have been tested and are out in the field.

There will be users who will resist the requirements that System Modelling enforces. Many Cedar programmers do not know or care which versions of the system they are using. These users can use "looser binding" to versions by omitting version information. However, use of this looser binding will not be recommended, since the version information helps detect version inconsistencies before a system is run. The experience with strong type-checking suggests some of our users will resist the extra discipline required by this approach.

*References To Files*

The environment in which the System Modeller runs was described in Chapter 2. The reader is referred to "Version and Size Problems" and "Single vs. Shared File Systems" in Chapter 2. There is one important difference between the environment in which the software that processes DF files was built and the environment in which the Modeller runs: The software described in this chapter makes use of the Cedar File System, which was not available when the

DF software was built. Use of the Cedar File System allows the System Modeller to manipulate files on remote servers when given file names that include host and directory information. Thus, the "BringOver" step of retrieving files to the local disk is not necessary. File names without a host or directory are assumed to be within a working directory. If a file is edited using the Cedar editor, the file system does not automatically store the new version on the same directory that contained the old version. The modeller will move such files when appropriate.

*File References in Models*

Models refer to files using an @-sign followed by a host, directory, and file name, optionally followed by version information. In a model, the expression

@[Indigo]<Cedar>X.Mesa!(July 25, 1982 16:10:09)

refers to the version of X.Mesa created on July 25, 1982 16:10:09 that is stored on [Indigo]<Cedar>. The !(...) is not part of the filename but is used to specify explicitly which version. The expression

@[Indigo]<Cedar>X.Bcd!(1AB3FBB462BD)

refers to the version of X.Bcd on [Indigo]<Cedar>X.Bcd that has a 48-bit version stamp "1AB3FBB462BD" (hexadecimal). For cases when the user wants the most recently-saved version of X.Mesa or X.Bcd,

@[Indigo]<Cedar>X.Mesa!H

refers to the most-recently stored version of X.Mesa on [Indigo]<Cedar>. This "!H" is a form of implicit parameterization. If a model containing such a reference is submitted as part of a Cedar release, this reference to the highest version is changed into a reference to a specific version.

## 4.3 User Interface

An interactive interface is provided for the modeller. When used interactively, the role of the Modeller is similar to that of an incremental compiler. The modeller tries to do as little work as it can as quickly as possible in order to produce a runnable system. To do this well, it works in parallel with the user, keeping track incrementally of as much information as possible about the objects under its control.

For example, consider the following scenario. Assume a model already exists and the user wants to change one module to fix a bug. Earlier, he has started the modeller with his working model as input. He uses the Cedar editor to make a change to a module. When the user finishes editing the module and creates a new version, the editor then notifies the modeller, indicating that a new version of the edited file now exists. If the source file being edited is referenced by the model, the modeller notices there is a new version and updates its description of the system

to refer to the new version. The user may edit and change more files. When he wants to make a version of his system, he issues another command to the modeller, which then compiles everything in correct order and (if there are no errors) produces an object file.

Another example would be when two programmers take a model and make their own versions, they may want to make a new version combining their changes. The modeller can provide help for this common case as follows: If one programmer has added, deleted, or changed some object (such as a module) in the model not changed by the other programmer, the modeller will add, delete, or change that object. If both programmers have changed the same object in different ways, the modeller cannot know which version to prefer and will ask the user for help.

At all points, a model is maintained describing the "current" system. When the user decides to release his system, he does so with an accurate description of the system in his model, thus minimizing software distribution errors. Since the models are simply text-files, the user always has the option of editing the model as he sees fit, so the modeller does not have to deal with obscure special cases that may arise.

*An Example*

The user begins by creating an "instance" of a modeller. An instance of the system modeller provides a window on the Cedar user's screen, as shown in Figure 4.1 at the end of this chapter. In this section we will give an overview of its use, suggested by the contents of Figure 4.1.

The modeller window is divided into four regions, which are, from top to bottom, 1) a set of buttons to control it, 2) a region containing fields where names may be typed, 3) a feedback area for compiler progress messages, and 4) a feedback area for modeller messages.

To help explain modeller operation, let us use a simple example and follow the steps the user performs to use the modeller.

**Step 1.** Assume that the modeller instance has just been created. The user decides to make changes to the modules in Example.Model. He enters the name of the model he is going to start with following the "ModelName:" field and pushes the "StartModel" button. From this point on the modeller is bound to Example.Model, and "StopModel" must be pushed before using the modeller on another model. "StartModel" initializes data structures in this instance of the modeller, "StopModel" frees the data.

**Step 2.** The user makes changes to files on his personal machine. The Cedar Editor has been modified to call the modeller to send a *Notice* operation to tell the modeller that a new version of a file exists. If the file being edited is in the model, the modeller updates its data structures to reflect the new version. If, for example, the user has added or deleted parameters, the modeller uses standard defaulting rules to modify the parameter list of the file in the model.

**Step 3.** Once he has made the intended edits, the user pushes "Begin," which a) recompiles modules as necessary, b) loads their object files into memory, and c) forks a process that starts the module executing. Modules will need to be recompiled if their corresponding source file has been edited or if any modules they depend on have been compiled. (Should a) or b) encounter

errors, the modeller does not proceed to c).)

**Step 4.** After testing his programs, the user may want to make changes simple enough that the old module may be replaced by the new module without re-loading and starting the system. If so, after editing modules, the user pushes "Continue," which tries to replace modules in the already-loaded system. If this succeeds, he can go on testing his program and the new code will be used. If the module is not replaceable, he must push "Begin," which will unload all the old modules in this model and load the new modules.

**Step 5.** After completing his changes, the user can push "StoreBack" to store copies of his files on remote file servers, and then push "UnLoad" to unload the modules previously loaded, and "StopModel" to free modeller data structures.

These steps are illustrated in Figure 4.2:

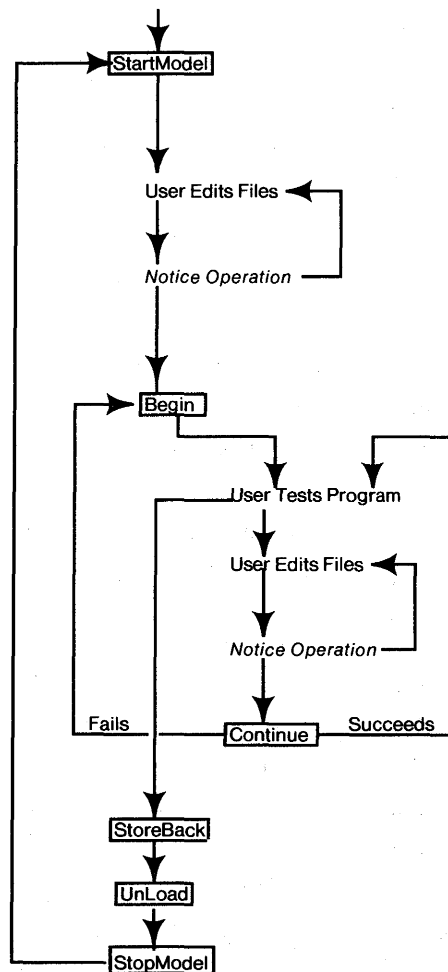



Figure 4.2 User Sequence

A program interface to take these steps is available and is described in Appendix C.

*Details*

**StartModelling:** The modeller begins by reading in the source text of a model and building an internal tree structure (described later) that is traversed by subsequent phases. These phases will use this tree to determine which modules must be compiled and loaded and in what order. Since parameters to files may have been defaulted, the modeller uses a database of information about the file to check its parameterization in the model and supply defaults, if necessary. If the database does not have an entry for the version of the file listed in the model, the modeller will read the file and analyze it, adding the parameterization information to the database for future reference. This database is described in Section 4.5.

**Notice Operation:** The Cedar editor notifies all modeller(s) running on the machine when a new version of a file is created. Each modeller searches its internal data structure for a reference to an earlier version of the file. If one is found, the modeller changes the internal data structure to refer to the new version.

While making edits to modules, users often alter the parameterization of modules (i.e., the interface types and IMPORTed interface records). Since editing the model whenever this happens is time-consuming, the modeller automatically adjusts the parameterization, whenever possible, by using the defaulting rules of the modelling language: If a parameter is added and there is a variable with the same name and type as the new parameter, that variable is used for the actual parameter. If a parameter is removed, then the corresponding actual is removed. The modeller re-parses the header of a "noticed" module to determine the parameters it takes.

Some changes made by the user cannot be handled using these rules. For example, if the user changes a module so that it imports an interface record, and there is no interface record in the model with that name, the modeller cannot know which interface record was intended. Similarly, if the user changes the module to export a new interface record, the modeller cannot know what name to give the exported record in the model. In these situations, the user must edit the model by hand to add this information and start the modeller again on the new version of the model.

**Compilation and Loading:** After the user pushes "Begin," the modeller uses the internal data structure as a description of a system the user wants to run on his machine. To run the system, each module must have been compiled, then loaded and initialized for execution. The modeller examines each module using the dependency graph implied by the internal data structure. Each module is compiled in correct compilation order if no suitable object file is available. Modules that take no parameters are examined first, then modules that depend on modules already analyzed are examined for possible recompilation, and so on, until, if necessary, all modules are compiled. Modules are only recompiled if 1) the modules they depend on have been recompiled, or 2) they were compiled with a different version of the compiler or different compiler switches than those specified in the model. If there are no errors, the modeller loads the modules by allocating memory for the global variables of each module and setting up links between modules by filling in the interface records declared in the module. When loading is completed, execution

begins.

**StoreBack:** Models refer to files stored on central file servers. The user types a file name without file server or directory information to the Cedar editor, such as "BTreeImpl.Mesa," and the editor uses information supplied by the modeller to add location information (file server and directory) for the files. (If the file name without location information is ambiguous, the user must give the entire file name to the editor.) To avoid filling file servers with excess versions, the modeller does not store a new version of a source file on a file server after the source file is edited. Instead, the new versions are saved on the local disk. When the user pushes "StoreBack" all source files that have been edited are saved on remote directories. A new version of the model is written to its remote directory, with references to the new versions of source files it mentions.

The compiler may have produced new versions of object files for source files listed in the model. Each object file so produced is stored on the same directory as its corresponding source file.

**Multiple Instances of Modellers:** More than one modeller may be in use on the same machine. The user can push the "NewModeller" button to create another window with the four subwindows described earlier. It is used in the same way as the modeller window described earlier. Two instances of a modeller can even model two versions of the same system model. Since file names without locations are likely to be ambiguous in this case, the user will have to type file names and locations to the editor and do the same for the "ModelName:" field in the modeller window.

### 4.4 Pragmatic Considerations in Implementation

The modeller must be able to analyze large collections of modules quickly, and must provide facilities normally associated with the loader, debugger, and other programs.

*Model Accelerators*

Some models are shared among many users, who refer to them in their own models by using the @-notation and then using returned values from these shared models. An example is the model "BasicCedar.Model," which returns a large number of commonly-used interfaces (interface types) that a Cedar user might use. Although it is always possible to analyze all sub-models such as BasicCedar.Model, retrieving the files needed for analysis is very time consuming.

When the user pushes "MakeModelBcd," the modeller makes an object file for a model, much as a compiler makes an object file for a source file. This model object file (called a *.modelBcd file*) is produced so that all parameters except interface records are given values, so it is a projection of the source file for the model and all non-interface record parameters. (This is analogous to the object files produced by the compiler, which are projections of the Cedar source

file and all the interface types, leaving the interface records as parameters.) The .modelBcd file acts as an accelerator, since it is always possible to work from the sources to derive the same result as is encoded in the .modelBcd. Its contents are described in section 4.6.

*Binding Functions*

The loading ability of the modeller gives the user the ability to load the object files of any valid model. This speed of loading is proportional to the size of the system being loaded and the inter-module references. As the system gets larger, it takes more time to load. However, the Cedar Binder has the ability to take the instructions and symbol table stored in each object file, merge these pieces of object, and produce an object file that contains all the information of the constituent modules while combining some tables used at runtime. This transformation resolves references from one module to another in the model, which reduces the time required to load the system and also saves space, both in the object file and when the modules are loaded. To speed loading of large systems, this feature has been preserved in the modeller. If "Bind" is pushed after "StartModel" and "Compile" or "Begin" are pushed, an object file with instructions and symbol tables merged is produced.

The programmer may choose to produce a bound object file for a model instead of a .modelBcd file when 1) the model is very large and loading takes too long or the compression described above is effective in reducing the size of the file or 2) the object file will be input to the program that makes the boot file for Cedar.

The use of bound object files is limited to a subset of all models because of restrictions imposed by the desire for compatibility with old object file formats, so its use is not encouraged if distributing a .modelBcd file is sufficient. Since this is how Cedar programs have been loaded in the past, the ability to bind code and symbols has eased our users conversion to the modeller.

*Module Replacement*

The ability to replace a module in an already loaded system can provide faster turnaround for small program changes. Module replacement in Cedar is possible if the following conditions are met:

1. The existing global data of the module being replaced may change in very restricted ways. Variables in the old global data must not change in position relative to other variables in the same file. New variables can only be added after the existing data. If the order changed, outstanding pointers to that data saved by other modules might be invalidated.
2. Any procedures that were EXPORTed by the old version of the module must also be EXPORTed by the new version, since the address of these objects could have been passed to other modules, e.g., a procedure that is passed as a parameter.
3. There are a number of architectural restrictions (such as the number of indices in certain tables) that must be obeyed.

4. No procedures from the affected module can be executing (or stopped as a breakpoint) the short period of time the replacement is occurring.

The modeller can easily provide module replacement since it loaded the modules initially and invokes the compiler on modules that have been changed. When the user pushes "Continue," the modeller tries to speed the compile-load-debug cycle by replacing modules in the system, if possible. Successful module replacement preserves the state of the system in which the replacement is performed.

The modeller calls the compiler through a procedural interface that returns a boolean *true* if rules 1 and 2 are obeyed; the modeller will also check that rules 3 and 4 are obeyed. If all four checks succeed, the modeller will change the runtime structures to use a new pointer to the instructions in the new module, which in effect replaces the old instructions by the new ones.

Some changes are substantial enough to violate rules 1-4, so after edits to a set of modules, some modules are replaceable and others are not. When this happens, the modules that are replaceable are replaced by new versions. The modules for which replacement failed are left undisturbed, with the old instructions still loaded. If desired, the user may try to debug those changes that were made to modules that were replaceable. If not, the user can push the "Begin" button to unload the current version and reload the system. Since no extra compilations are required by this approach, the user will always try module replacement if there is a possibility it will succeed and he wants to preserve the current state of the program. There is no time penalty if module replacement fails.

*Debugger Interface*

When the Cedar debugger examines a stopped system (e.g., at a breakpoint) the debugger can follow the procedure call stack and find the global variables for the module in which the procedure is declared (these global variables are stored in the *global frame*). The modeller can provide the debugger with module-level information about the model in which this module appears, and provide file location and version information. This is particularly useful when the debugger wants to inspect the symbol table for a module, and the symbol table is stored in another file that is not on the local disk.

The programmer deals with the model naturally while debugging his system. The modeller provides an interface (called *RTModel*, described in Appendix C) called by the debugger as it needs module-level information.

Since more than one modeller can be in use on a machine, the modeller(s) call procedures in an independent runtime loader to add each model to a list of models maintained for the entire running system. When the modules of a model are loaded or unloaded, this list is updated, as appropriate. To simplify the design, the list of models is represented by the internal data structures used by the modeller to describe a model. This model has no formal parameters and no file where it is stored in text form, but it can be printed. This allows the debugger to use a simple notion of scope: a local frame is contained in the global frame of a module. This module is listed in a model, which may be part of another model that invokes it, and so on, until this

top-most model is encountered. The debugger can easily enumerate the siblings in this containment tree. It can enumerate the procedures in a module, or all the other modules in this model, as appropriate. This type of enumeration occurs when the debugger tries to match the name of a module typed by the user against the set of modules that are loaded (e.g., to set the naming environment for expressions typed to the debugger).

### 4.5 Data Structures and Tables Used

The procedures of the modeller can be categorized into these functional groups:

1. Procedures to parse model source files and build an internal parse tree.
2. Procedures to parse Cedar source and object files to determine needed parameterization.
3. Procedures that maintain a table (called the *projection table*) that expresses relationships between object files and source files, as described below.
4. Procedures that maintain a table (called the *file type table*) that gives information about files described in models. This includes information about the parameters needed by the file (e.g., interface types) and information about its location on the file system.
5. Procedures that load modules and maintain the top-level model used by the debugger.
6. Procedures used to call the compiler, connect the modeller to the editor, and other utility procedures.
7. Procedures to maintain version maps.

The sections below discuss essential internal data structures used in these groups, which are shown in Figure 4.3.

*Internal Parse Tree*

The model is read in from a text file and must be processed. The modeller parses the source text and builds an internal parse tree. This parse tree has leaves reserved for information that may be computed by the modeller when compiling or loading information. When a *Notice* operation is given to the modeller, it alters the internal data structures to refer to new versions of files. Since new models are derived from old models when *Notice* operations occur, the modeller must be able to write a new copy of the model it is working on. When a source file for the new model is written out, the information in the internal parse tree is used, in a pretty-printed form.

There is one parse tree per source model file. The links between model files that are "called" by other model files are represented as pointers from one model's internal data structure to another in virtual memory. (Procedures that traverse these internal data structures for one model may or may not follow these pointers to other models.)

The internal data structure represents the dependency graph used to compile modules in correct compilation order by threading pointers from one file name to another in the parse tree.

*Model-Independent Tables*

Three tables are maintained independently from instances of the modeller on a machine. These tables serve as accelerators for the modeller and are stored as files on the local disk.

The information can be automatically reconstructed whenever it is not present; as a result, the information is never purged. When the file containing the table becomes too large the user simply deletes it from his local disk, and the information is reconstructed (as described at the end of the next section).

**File Type Table:** This table contains a list of files that are referenced by models and have been analyzed. The modeller abstracts essential properties of the files in models and stores the information in this table. For example, a Cedar source file is listed along with the implied procedure type used by the modeller to compile and load it. The file type table also contains information that records whether a file has been edited, and if so, whether it has been saved on a remote file server.

**Projection Table:** This table keeps a list of entries that describe the results of running the compiler (or other programs) that take a file and parameters the file needs (such as interfaces) and produce object files. Before invoking, for example, the compiler on a source file to produce an object file, the modeller consults this table to see if such a file is already available. If an entry is not in the table, there may be an object file on the disk made by the compiler that predates the information in the projection table. If not, the compiler is invoked to produce the object file. In either case a new entry is added to the table for later use. The projection table does not include the location of object files. Version maps, described below, are used for this.

**Version Maps:** The central file servers used by the system modeller can store more than one version of a source file in a directory. Each version is given a version number, which ranges from 1 to 32767 and is typically less than 100. Obtaining the creation time (of a source file) or the 48-bit version stamp (of Cedar object files) from a central file server takes between 1/4- and 1-second. For directories with many versions of a file, searching for the create time (or version stamp) can take a few seconds *per file.*

Since the modeller must determine the explicit version number of the file that is referenced in the model, this slow search for large numbers of files referenced by models is prohibitively expensive. To avoid this excessive searching when it is running, the modeller uses an index between create times (or version stamps) and full path names that include explicit version numbers for files. Since the version numbers used by the file servers are not unique and may be re-used, the modeller uses this index as a cache of hints that are checked when data in the file is actually used. If there is no entry for a file in the cache, or if it is no longer valid, the versions of a file are searched and an entry is added or updated if already present. Commonly-referenced files of the Cedar system are inserted in a version map maintained on each machine.

These tables are illustrated in Figure 4.3 at the end of this chapter.

In summary, the File Type table speeds the analysis of files, the Projection table speeds the translation of objects into derived objects, and Version Maps are used to avoid extensive directory searches.

### 4.6 Interaction Between Tables and .modelBcd files

*Contents of .modelBcd Files*

A .modelBcd file can be produced for a model that has been analyzed by pushing the "MakeModelBcd" button. The .modelBcd file contains the same information described in the previous tables. Only information relevant to the model being is analyzed is stored. The .modelBcd contains a) a representation of the internal parse tree that results from reading and parsing the source file for the model, b) a file type table for source files referenced by the model, c) a projection table describing the object files that are produced, for example, by the compiler, and d) a version map that describes, for each source and object file in b) and c), a file location including a version number.

A model may refer to other models in the same way it refers to other Cedar source files. The projection table includes references to .modelBcd files for these inner models.

*Use of this Information*

The information stored in the model-independent tables or present in .modelBcd files is used in four different ways: three ways when the modeller is used, and once by the release process (described in the next section).

**StartModelling Analysis:** Each application of a source file to a parameter list in the model is checked for accuracy and to see if any parameters have been defaulted. The version information (create time) following the source file name is used to look up the parameters needed by the file in the file type table. If no entry is present, the source file must be parsed to get its parameters. The version map is used to obtain an explicit file on a file server. If there is no entry for the create time of this file in a version map, all versions of the source file on the directory listed in the model are examined to see if they have the right create time. If so, an entry for that version is added to the version map and the file is read and its type is added to the file type table. If no such version can be found by enumeration, an error is reported.

If the version of the source file is given as "!H", meaning the highest version on that directory, the directory is probed for the create time of the highest version, and that create time is used as if it were given instead of "!H".

Figure 4.4 shows how a reference to "[Ivy]<Schmidt>X.Mesa" of July 25, 1982 14:03:02 is treated by the StartModelling analysis.

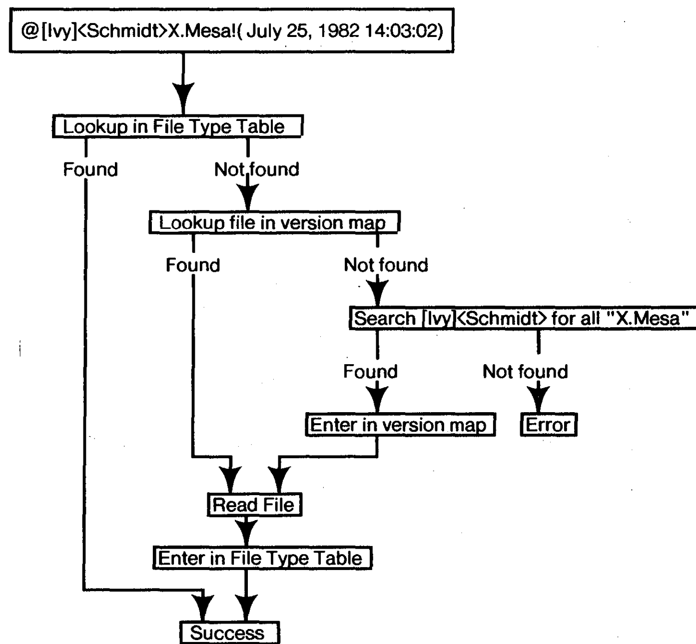



Figure 4.4 StartModelling Analysis

**Compilation Analysis:** After the user pushes "Begin" or "Compile", the modeller constructs object files for each source file in the model. Each source file and its parameters is looked up in the projection table. If not present, the modeller constructs the 48-bit version stamp that an object file would have if it had been compiled from the source and parameters given. The version map is used to search for an object file with this 48-bit version stamp. If not found in the version map, the modeller searches for an object file in the directory where the source file is stored. If found, an entry is added to the version map and to the projection table.

The modeller does not search for object files compiled from source files that have just been edited since it knows these have to be compiled.

If the modeller must compile a source file because it cannot find an object file previously compiled, the source file is read using the version map entry for the source and an object file produced on the local disk. Information about this object file is added to the model-independent tables and version maps. The object file is stored on a file server later when "StoreBack" is pushed. (See figure 4.5)

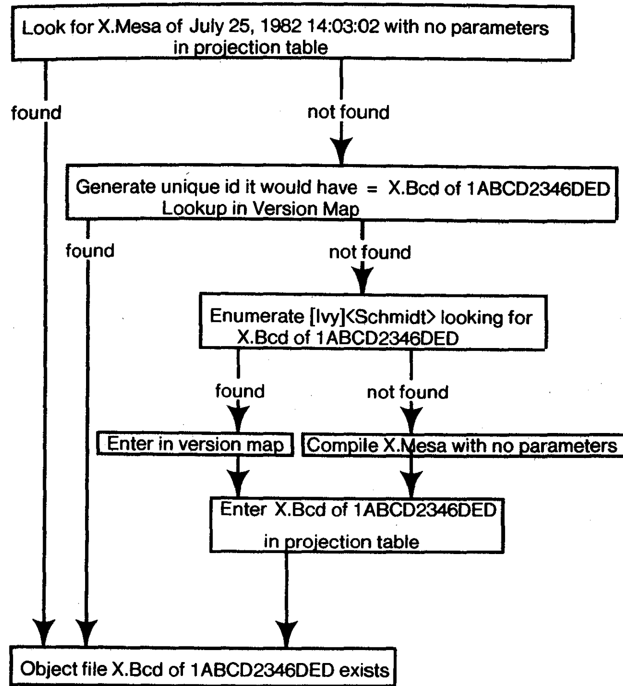



Figure 4.5 Compilation Analysis

**Loader Analysis:** Each object file must be read to copy the object instructions into memory. The modeller loader looks up the 48-bit version stamp in the version map to find the explicit version of the file to read. (See figure 4.6)

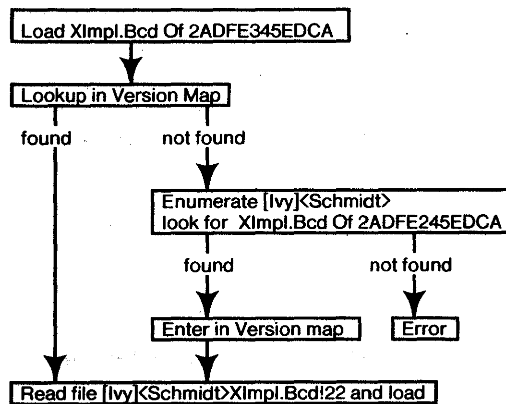



Figure 4.6 Loading Analysis

Since the version maps are hints, the presence of an entry for a file in a version map does not guarantee that the file is actually present on the file server, therefore, each successful probe to

the version map delays the discovery of a missing file. For example, the fact that a source file does not exist may not be discovered until the compilation phase, when the modeller tries to compile it. This means the modeller has to be robust in the face of such errors. For released software (see below), we guarantee the files are present.

*Retention of Information in Tables*

When the modeller stores file type, projection, and version map information in .modelBcd files, it stores only information relevant to the model in use. When the modeller reads .modelBcd files, it takes the information from the .modelBcd and adds it to tables maintained on each personal computer. When a module is compiled for the first time, this information is added to the tables managed centrally on each computer. This information can, over time, become obsolete and require large amounts of disk space, since these tables are stored in files on the local disk. If these files are deleted from the local disk, the modeller will reconstruct the information as it uses it. Some of the information is also stored in .modelBcd files, so, in many cases, little extra work is required to reconstruct this information.

## 4.7 Releases of Cedar using the Modeller

Use of the modeller will eventually replace use of DF files in Cedar. As described in detail in Chapter 2, Cedar releases are currently done using DF files. This section describes how releases will work when they involve models instead of DF files. The last subsection in this section compares the current approach using DF files with the one detailed below.

Some aspects of the release process will not change. We anticipate the same frequency of releases when they are model-based. The pre-release activities of the Release Master described in Chapter 2 will not change.

*The Working Position Model*

The Release Master maintains a model that is a list of model objects. This list (called the *working position model*) defines, for every model named in the list, a file server and directory where it can be found. While a release is being developed, this model refers to objects on their working directories; for example, the working position model might contain

```
Top ~ [
    BTreeModel ~ @[Indigo]<Int>BTree.Model!H -- ReleaseAs [Indigo]<Cedar>--,
    RuntimeModel ~ @[Indigo]<Int>Runtime.Model!H -- ReleaseAs [Indigo]<Cedar>--
    ]
```

The working position model is used during the development phase as a description of models that will be in the release, and gives file locations of these files while they are being developed. The working position model provides the list of models that will be released. Models not mentioned in the working position model will not be released. Note that two versions of the

same model can be released as long as they have different names in the working position model.

*Release Mechanics - User*

Every model being released is expected to have a LET statement at the beginning that uses the working position model to define the objects declared in the model. The model is expected to use the defined objects to refer to other models.

```
LET @[Indigo]<Int>WorkingPositions.Model!H IN [
    ...
    [RTTypes: INTERFACE] ~ RuntimeModel[],
    ....
    ]
```

Users are not allowed to refer to models by @-reference since there is no way to tell if those models are being released. They may only refer to models listed in the working position model.

*Release Mechanics - Implementor*

Models being released must also have a comment that contains the object name in the working position model (e.g., "BTreeModel") and the working directory that has a copy of the model, e.g.,

```
-- ReleaseName BTreeModel
-- WorkingModelOn [Indigo]<Int>BTree.Model
```

The model must declare the release position of each file by appending the release position of the file as a comment after the filename in the model, e.g.,

```
@[Ivy]<Work>XImpl.Mesa!H (-- ReleaseAs [Indigo]<Cedar>XPack>--)[ ]
```

A default ReleaseAs comment can define the release position of files in the model (which may differ from the release position of the model), for example, if the model contains a comment

```
-- DefaultReleaseAs [Indigo]<Cedar>BTrees>
```

then the user may omit the "-- ReleaseAs [Indigo]<Cedar>BTrees> --" clauses.

*Non-Program Files*

We encourage users to add relevant non-program files (such as files containing documentation) to models. Models refer to such files the same way they refer to program files. Each such expression has type

([] → [STRING])

and returns the file's name as a string. The Release Tool moves these files to their release position as it moves any other file. For example,

dontCare: STRING ~ @[Ivy]<Schmidt>Doc.Tioga!H (-- ReleaseAs [Indigo]<Cedar>Docn>--)[ ]

in a model ensures that the file "Doc.Tioga" will be stored on the Cedar documentation directory. The name "dontCare" is bound to the resulting string, which can be ignored.

*The Model Release Tool*

After all models that will be released have been prepared, the Release Master runs the Release Tool, which makes three passes over the models being released.

*Phase One: Check*

The check phase of the Release Tool checks the working models for problems that might prevent a successful release. This phase checks that all objects referred to by the working model exist. It also checks that derived objects (such as .Bcd files) exist. This guards against compilation errors in the source files.

Each model is parsed and all files listed in the model are checked. Phase one ensures that the versions listed in the models exist and checks that their parameterization is correct. The directory containing each source file is checked to make sure it contains a valid object file. Common blunders, such as a reference to a model that is not in the working position model, are caught. The Release Master contacts implementors and asks them to fix any errors caught in this phase.

The checking of parameterization and object existence is all we can do to test the release automatically. Since the check phase always finds a few mistakes the first time it is run, phase one can be repeated a few times until these errors are eliminated.

*Phase Two: Move*

The move phase moves the files of the release onto the release directory and makes new versions of the models that refer to files on the release directory instead of the working directory. For each model listed in the release position list, the move phase

1. reads in the model from the working directory,
2. moves all files explicitly mentioned in the model to their release position, and
3. writes a new version of the source file for the model in the release directory.

This release version of the model is like the working version except that a) all working directory paths have been replaced by paths on the release directory, b) a comment is added recording the working directory that contained the working version of the model, and c) the LET statement referring to the release position list is switched to refer to the one on the release directory. The model may look like this:

```
-- ReleaseName  BTreeModel
-- CameFromModelOn  [Indigo]<Int>BTree.Model
-- DefaultCameFrom  [Indigo]<Int>BTrees>
LET  @[ivy]<Rel>ReleasePosition.Model  IN  [
     ...
     RTTypes: INTERFACE  ~  @[Indigo]<Cedar>XPack>file.bcd!1234
            -- CameFrom  [Indigo]<Int>XPack>--,
     ...
     ]
```

Any references to highest version ("!H") are changed to be explicit create times as the model is written.

At the end of phase two, the working position model is automatically converted to a *release position model* that defines the same variables as the working position model, but sets those variables to refer to the model stored on the release directory. A release position model might be

```
Position  ~  [
     BTreeModel  ~  @[Indigo]<Cedar>BTree.Model!2344,
     RuntimeModel  ~  @[Indigo]<Cedar>Runtime.Model!2345]
     ]
```

Note that the LET switch is a deviation from explicit parameterization that allows us to change the nature of each model from being a development version to being a released version. The LET switch could be avoided if every model took a parameter that controlled whether its LET statement should refer to the working position model or the release position model. The SML language could be augmented with a type "BOOLEAN" and an IF-THEN-ELSE expression to accomplish this. Because the Release Tool has to rewrite models anyway to eliminate "!H" references, we chose to provide the LET switch automatically.

Phase two also constructs a directed graph (that must be acyclic) of models in reverse dependency order that will be used in phase three. In this dependency graph, if model A refers to model B, then B has an edge to A.

Figure 4.7 shows the movement of files by this phase.

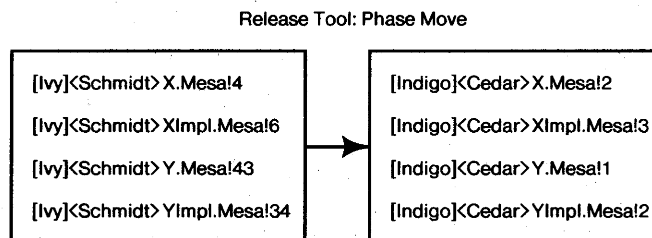



Figure 4.7

*Phase Three: Build*

The build phase takes the dependency graph computed during the move phase and uses it to traverse all the models in the release. For each model:

1. All models on incoming edges must have been examined.
2. For every source file in the model, its object file is moved to the release directory from the working directory.
3. A .modelBcd file is made for the version of the model on the release directory.
4. If a special comment in the model is given, a fully-bound object file is produced for the model (usually to use as a boot file).

After this is done for every model, a version map of the entire release is stored on the release directory.

Figure 4.8 shows the movement of files by this phase.

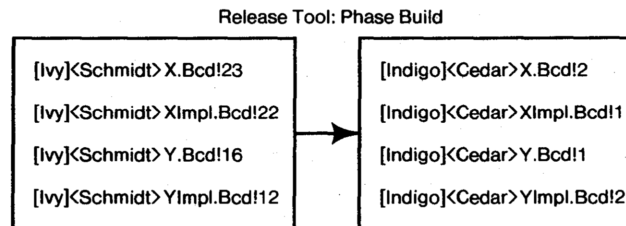



Figure 4.8

At the conclusion of phases check, move, and build, we have established that

1. (Check) All reachable objects exist, and derived objects for all but the top object have been computed. This means the files input to the release are statically correct.
2. (Move) All objects are on the release directory. All references to files in these models are by explicit create time (for source files) or version stamps (for object files).
3. (Build) The system has been built and is ready for execution. All desired accelerators are made (.modelBcd files and a version map for the entire release).

*Phase Implementation Details*

**Phase Check.** In order to know the parameterization of files referenced in the model, some part of each Cedar file must be read and parsed. Because of the large number of files involved, phase one maintains file type and projection tables and a version map for all the files on their working directories. These tables are filled by extracting the tables stored in the .modelBcd files for the models being submitted to the release. Any models without .modelBcd accelerators are read last in phase one and the result of analyzing each file is entered into the database. The version map information about object file location(s) and projection table are used later in phase three.

Because files can be deleted by mistake after the .modelBcd file is made and before phase one is run, the Release Tool checks that every version of every file in the release is present on the file server by verifying the file location hints from the .modelBcd files. This is the most time-consuming part of the check phase.

**Phases Move and Build.** The move and build phases could have been combined into a single phase. Separating them encourages the view that the build phase is not logically necessary, since any programmer can build a running Cedar system using the source models and Cedar source files that are moved to the release directory during the move phase. The build phase makes a runnable system once for all users and stores the object files on the release directory.

The build phase could be done incrementally, as each model is used for the first time after a release. This would be useful when a release included models that have parameters that are unbound, which requires the user to build the model when the model is used and its parameter are given values.

The check phase file type and projection tables and version map are used to make production of the .modelBcd files faster. The projection table is used to compute the version stamps of object files needed, and the version map is used to get the filename of the object file. This object file is then copied to the release directory. The file type entry, projection entry and new release position of source and object files are recorded in the .modelBcd being built for the released model.

The build phase has enough information to compile source files if no suitable object files exist (in step 2 of phase three). To speed up releases, we require that the programmer make valid object files before we run phases two and three. If such an object file is not on the same directory as the source file, we notify the programmer of his error and ask him to prepare one. If the Release Master ran the compiler, he would most likely compile a file that the programmer had forgotten to recompile, and this file might have compilation errors in it. The ability to automatically compile every file during a release is useful in extensive bootstraps, however. For example, a conversion to a new instruction set, where every module in the release must be compiled, is easily completed using a cross-compiler during phase three.

The build phase produces the version map of the release by recording the create time or version stamp of every file stored by the Release Tool on the release directory, along with file server, directory, and version number for the file. The version maps supplied by the .modelBcd files that were submitted to the release cannot be used, since they refer to files on their development directories and not the release directories. This released version map is distributed to every personal machine. Although the .modelBcd files also have this information, it is convenient to have it all in one map.

Figure 4.9 has an example of this version map.

Version Map After Release

| Index | File Location |
|---|---|
| X.Mesa of July 25, 1982 14:03:02 | [Indigo]<Cedar>X.Mesa!2 |
| XImpl.Mesa of July 25, 1982 14:05:06 | [Indigo]<Cedar>XImpl.Mesa!3 |
| Y.Mesa of July 25, 1982 15:05:08 | [Indigo]<Cedar>Y.Mesa!1 |
| YImpl.Mesa of July 25, 1982 15:07:03 | [Indigo]<Cedar>YImpl.Mesa!2 |
| X.Bcd of 1ABCD2346DED | [Indigo]<Cedar>X.Bcd!2 |
| XImpl.Bcd of 2ADFE345EDCA | [Indigo]<Cedar>XImpl.Bcd!1 |
| Y.Bcd of 3421ABD4235A | [Indigo]<Cedar>Y.Bcd!1 |
| YImpl.Bcd of 23455BBDC63B | [Indigo]<Cedar>YImpl.Bcd!1 |

Figure 4.9

*Multiple Levels Of Models*

The working position model may list other nested working position models. The objects defined in the nested working position model are named by qualifying the name of the outer object. For example, if Top contained

```
Top ~ [
    ...
    NestedSet ~ @[Indigo]<Int>NestedWPM.Model!H -- ReleaseAs [Indigo]<Cedar> --,
    ...
    ]
```

Then, the elements of the nested working position model can be referred to using "." notation, e.g., Top.NestedSet.Element. The ReleaseAs clause in Top indicates the directory in which the analogous release position model is written. The same algorithm is used to translate the working model into a release model.

*Release Comparison*

We have had extensive experience making releases based on DF files as input. There are a few differences between the release procedures and algorithms used for DF files and those proposed here for models.

**The Working Position Model-Release Position Model.** The DF Release Tool works from a DF file that serves the same function as the working and release position models. DF files, however, have no provision for parameterization, so the Release Tool changes all references to DF files that are on working directories to references to newer versions in their release directories. However, the implementor must be told the name of the working directory for the DF file, which may change between releases. The use of one model that defines where the models are, and that is used to control which modules are being released, allows the programmer to use an object value

that may change without concern about new or different versions. The DF software depends on the fact there is only one version of each DF file being released each time. The use of the Working Position Model forces the models to refer explicitly to the objects of the models they want. This allows the Working Position Model to refer to two different versions of a model by using different names for their objects.

**VerifyDF is Eliminated.** VerifyDF analyzes object files to determine the module interconnection structure and to verify that consistent versions were used to make the object files listed in the DF file. Models are used in a more "active" way: the interconnection structure is explicit in models, which are statements of what is desired. If object files are inconsistent, the modeller has enough information to recreate them. VerifyDF also checks for omissions of files needed by a DF file to describe a package. In models, this corresponds to an undefined variable or parameter that is caught when the model is parsed.

**Difference in Phases.** There are three phases in each release algorithm, but they are used differently. Phases one and two of the DF Release Tool perform consistency and completeness checking, performed in the check phase of the Model Release Tool. Phase three of the DF Release Tool moves all files, including both source and object files. The move and build phases of the Model Release Tool move source and object files to the release directory, respectively.

**Model Approach is More Powerful.** DF files do not have enough information to build systems automatically. When there is a problem with the input to the DF release process, the Release Master must wait for the implementors to make changes. The Model Release Tool has the ability to run the compiler, although we plan to use this facility sparingly. It is essential for bootstraps of the entire system where all modules need to be compiled in correct dependency order.

### 4.8 Extensions To This Approach

*Sharing of Models*

During the development of Cedar, most packages of logically-related software have been maintained by one person. Whenever two people have had to maintain separate versions, each has made a copy of the model describing the package and has stored it in a personal directory. When the versions were merged back into a new, common version of the model, the programmers have edited the model by hand.

The Cedar project is small enough that this solution is sufficient for our needs. In larger projects, especially those where the programmers do not work near each other, more formal "forked" development facilities are required. Since models are text files, information could be added to models to uniquely identify a common ancestor at the time versions are forked. (This is essentially the technique used in SCCS [Glasser, 1978], except all versions are stored in one file and this approach makes separate copies.) When two versions are merged, an automatic tool could

merge them according to the following rule: (see also figure 4.10)

Consider a common ancestor model A and two descendents S1 and S2 that were derived from A, some time ago. A new A' is to be derived by merging S1 and S2. For each object defined in S1 and S2: if there is a new version of the object in S1 and it is unchanged in S2, add the new version to A'. If there is a new version of the object in S2 and it is unchanged in S1, add the new version from S2 to A'. If the object was not changed in either S1 or S2, use the version described in A. If the object was changed in both S1 and S2, then the merge facility cannot decide which of the two versions to choose and asks the programmer(s) which version to use in A. If the object is a model, the procedure recurses and tries to produce a new A' for the object.

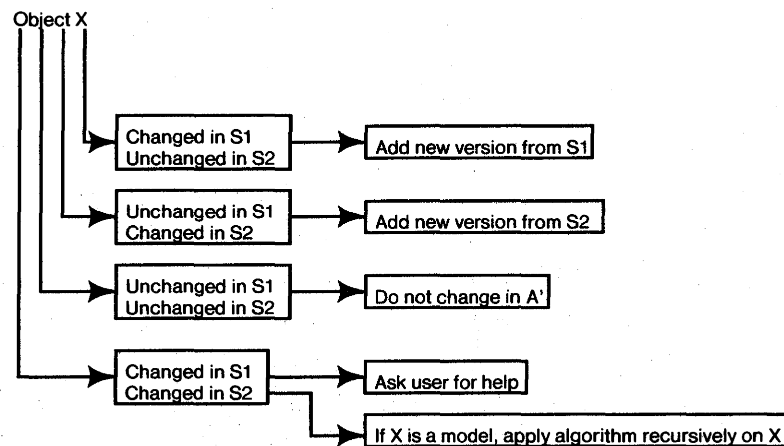



Figure 4.10 Merge Algorithm

Although the algorithm above is not foolproof, it is probably the best automatic way to handle merging of versions since, in the most common cases, the newer version is the one intended. If there are two newer versions, there is no way to automatically choose without some other information from the user.

*Use of a Database*

The information maintained in tables on the local disk, in version maps, and in .modelBcd files could be stored in a database of program information that is centrally-maintained and accessible by each personal machine over the network. Since this database would contain information about many different models and modules being developed, it would store information in one place that, in a system without a database, would be expensive to compute. Query facilities that might be available in the database system could easily be used to answer questions about the programs under development that require specialized analysis programs if no database is used, such as "who depends on module X?"

If the database were used to replace the .modelBcd file and tables maintained by the modeller, the database would have to handle concurrent queries and updates to its information, generated by modellers that are running. It would also have to answer queries quickly enough that the time required to answer the query and transmit the information would not be much greater than the time to search specialized tables on the local machines. If this is not feasible, a central database could be used as a cache of information duplicated in .modelBcd files and in the tables on the local disk. Instances of modellers that are running could periodically update the database so that queries from other programs or queries made by users worked from a database of current information. However, with any centrally-managed database of programs, there is a problem of deleting old information. Our current approach stores the information in files that must periodically be deleted as versions of the Cedar system become obsolete.

It would be possible to store models directly in a database, using links to represent dependency or interconnection information now expressed as programming language statements stored in a text file for the model. The direct representation of models in a database would allow more flexibility in version references than is available using models, since changes to the database would not be rigidly hierarchical, as they are with models: if a file in a model is changed, a new version of that model is created. If another model refers to the old version of this model, a new version of this outer model must also be created, etc.

We believe use of explicit version references promotes better organization on the part of programmers working in a large system. In our scheme, programmers are forced to view any change to a module in terms of the change to the package(s) that contain it. No change can go unnoticed when all files and all models that contain them are referenced with explicit unique-ids. System models are similar enough to Cedar programs that their use is familiar to implementors. A more flexible representation offers uncertainty that we must constrain in a multi-person project, especially when the components submitted to a release are not completely constrained by version as they are when the system refers to models.

Future plans include integration of the System Modelling databases with the Cedar database system. Databases will almost certainly be used to store both Cedar modules and system models in the future. (This is discussed in the "Future Research" section of Chapter 5.)

## 4.9 Current Status

*Conversion To Full Use*

At present, the system described above is not completely integrated into the Cedar system. It is expected to be completed over the next year. Although some users are using the modeller to compile their systems, most use manual techniques. The existing modeller has been used by five or six programmers over the last year. Two programmers have used it heavily. All the functional groups listed in section 4.5 (except Version Maps) have been implemented and have been heavily

tested. In particular, the user interface, module analysis, module replacement, and module loading code has been extensively tested. The language parsed by the modeller differs slightly from the one in Chapter 3; the modeller is being converted to the newer syntax. The existing file system code is being replaced by code that calls the Cedar File System, now that it is ready. Code to produce .modelBcd files, the debugger interface, and the Model Release Tool are not implemented yet.

At present, all Cedar programmers use DF files [see Chapter 2] to describe the files that are part of this system. Since each DF file lists the files needed to compile modules in a package or program, DF files provide the file information part of the information system modelling manages. We envision two stages in the integration of the modeller in the system and replacement of the DF software in Cedar.

**Stage One.** Remaining functionality will be provided, and all users will be encouraged to use the modeller. This involves implementing the code needed to produce .modelBcd files, code to produce fully-bound models (for the "Bind" button), integration with the debugger, and code to call software to make and use version maps. We anticipate a long period of performance tuning and maintenance while users begin to use the modeller. Users will still use DF files to describe release submissions. We may implement a program that produces a DF file from a model, if there is sufficient interest. Since the file information is present in the model, this will be easy.

**Stage Two.** At some point, most users will have converted to use of the modeller and we will perform a release driven by models, instead of by DF files. Based on experience building the DF Release Tool, the first releases using models will be difficult to make, and users will experience some delays while we are fixing problems. The Cedar system is now very large and we estimate that one or two working days will be required to run phases check, move, and build of the modeller release tool. To run without failure for many hours, the release tool must be able to recover gracefully from temporary resource limits, such as file servers being down or too busy for additional use, and directories that are full.

Because DF files do not contain as much information as models, we cannot make a release where some components are described by DF files and others are described by models.

### 4.10 Conclusion

We have described a program development tool that automates part of the compile-edit-debug cycle. This tool, called the System Modeller, uses system models as descriptions of software systems. System models combine, in one place, the three kinds of information that must be managed: file version information, compilation information, and interconnection information.

To achieve acceptable real-time performance, this tool uses specialized databases that have information about the files that are part of the system being analyzed.

A proposal has been made for extensions to the syntax of models that will allow automated releases of consistent Cedar systems. These releases will have the highest degree of consistency of software because the version stamps of interfaces used will be rigorously checked.

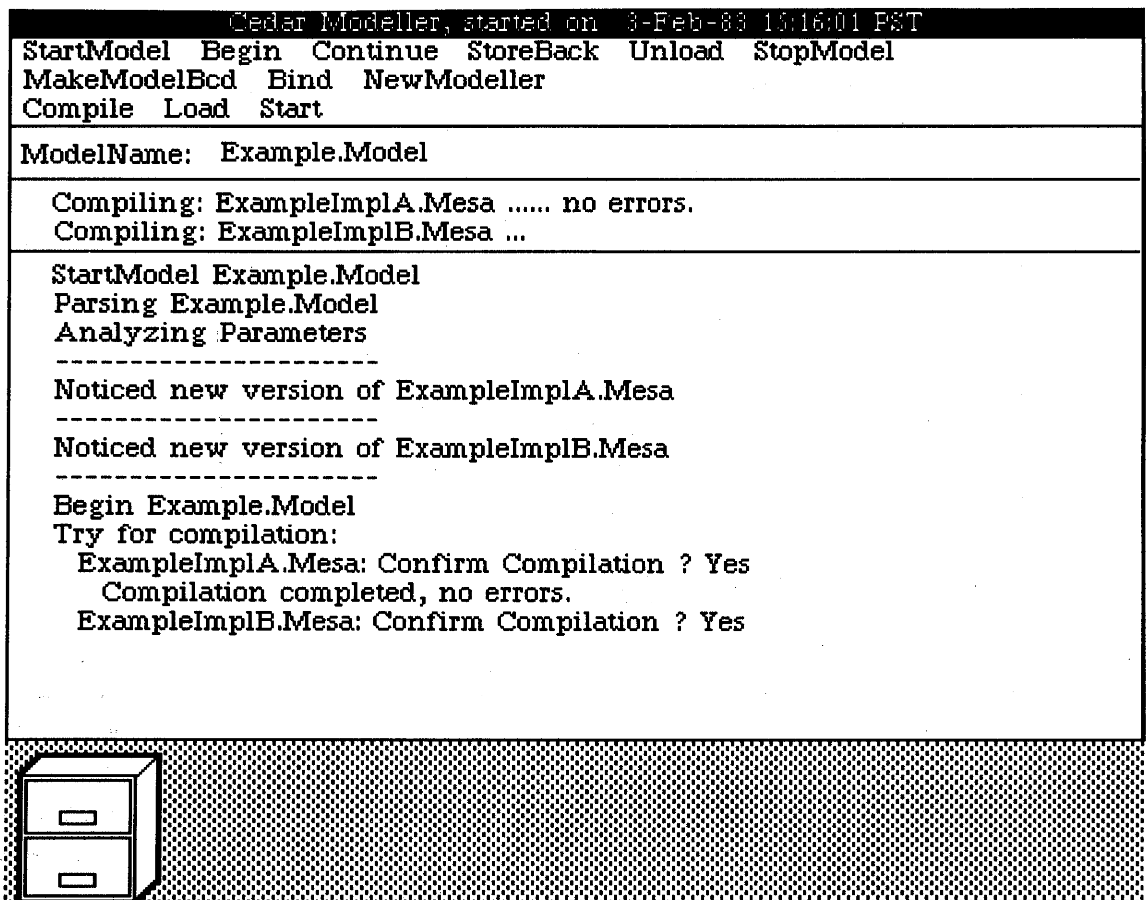



Figure 4.1

Example

```
-- DefaultReleaseAs [Indigo]<Cedar>
[X: TYPE X ~ @[Ivy]<Schmidt>X.Mesa!(July 25, 1982 14:03:02)[],
 XImpl: X ~     @[Ivy]<Schmidt>XImpl.Mesa!(July 25, 1982 14:05:06)[X],
 Y: TYPE Y ~  @[Ivy]<Schmidt>Y.Mesa!(July 25, 1982 15:05:08)[],
 YImpl: Y ~     @[Ivy]<Schmidt>YImpl.Mesa!(July 25, 1982 15:07:03)[X, Y, XImpl]
]
```

File Type Table

| Source Object | Type |
|---|---|
| X.Mesa of July 25, 1982 14:03:02 | [] -> [TYPE X] |
| XImpl.Mesa of July 25, 1982 14:05:06 | [X: TYPE X] -> [XImpl: X] |
| Y.Mesa of July 25, 1982 15:05:08 | [] -> [TYPE Y] |
| YImpl.Mesa of July 25, 1982 15:07:03 | [X: TYPE X, Y: TYPE Y, XImpl:X] -> [YImpl: Y] |

Projection Table

| Source Object | Parameter Values | Result Object |
|---|---|---|
| X.Mesa of July 25, 1982 14:03:02 | [] | X.Bcd of 1ABCD2346DED |
| XImpl.Mesa of July 25, 1982 14:05:06 | [X.Bcd of 1ABCD2346DED] | XImpl.Bcd of 2ADFE345EDCA |
| Y.Mesa of July 25, 1982 15:05:08 | [] | Y.Bcd of3421ABD4235A |
| YImpl.Mesa of July 25, 1982 15:07:03 | [ X.Bcd of1ABCD2346DED, Y.Bcd of3421ABD4235A, Y.Bcd of3421ABD4235A ] | YImpl.Bcd of 23455BBDC63B |

Version Map

| Index | File Location |
|---|---|
| X.Mesa of July 25, 1982 14:03:02 | [Ivy]<Schmidt>X.Mesa!4 |
| XImpl.Mesa of July 25, 1982 14:05:06 | [Ivy]<Schmidt>XImpl.Mesa!6 |
| Y.Mesa of July 25, 1982 15:05:08 | [Ivy]<Schmidt>Y.Mesa!43 |
| YImpl.Mesa of July 25, 1982 15:07:03 | [Ivy]<Schmidt>YImpl.Mesa!34 |
| X.Bcd of 1ABCD2346DED | [Ivy]<Schmidt>X.Bcd!23 |
| XImpl.Bcd of 2ADFE345EDCA | [Ivy]<Schmidt>XImpl.Bcd!22 |
| Y.Bcd of 3421ABD4235A | [Ivy]<Schmidt>Y.Bcd!16 |
| YImpl.Bcd of 23455BBDC63B | [Ivy]<Schmidt>YImpl.Bcd!12 |

Figure 4.3 Tables

# 5. Conclusion

In Chapter 1, we introduced the three kinds of information that must be managed: file information, compilation information, and version information. This thesis has shown that these kinds of information can be managed by a combination of 1) new description mechanisms, such as DF files and system models, 2) tools to automate the production of software, such as the Modeller, BringOver, etc., and 3) managerial techniques, such as the release process. This thesis has shown that there are solutions suitable for small and large systems being developed in a distributed environment. These program development tools are fast enough to be used by programmers in normal software development.

The DF software and Release Process have solved the version control problem encountered while building the Cedar system. The DF system has proved popular for both implementors of Cedar and individual programmers using Cedar. Beginning users of Cedar now learn to use DF files as soon as they have begun building their first Cedar program. Our extensive experience (fifteen releases) with (at present) 4700 files and 447,000 lines of code shows that this is a "production-level" system that has stood the test of real use (see Appendix A). The DF software handles a DF file as large as Runtime.DF in Appendix B, without complaint. As noted in Chapter 3, the DF system is now being used in another division of Xerox, as well. We conclude that a system devoted to managing versions in our environment with a simple language (DF files) to describe systems can handle a large software system without placing undue burdens on the programmers or the Release Master.

We have less experience with system models and the system modeller. The prototype modeller works as described in Chapter 4 but has had many performance problems. The use of caches of data makes the modeller possible. Without them, the time required to process information about Cedar systems is too great. The automatic recompilation and module replacement facilities of the modeller have been very popular, as expected. The underlying complexities of Cedar's module interconnection facilities can make system models very hard to understand, and the size of Cedar systems makes the algorithms that manipulate this information more complex.

We conclude that system models have all the information needed by a program management tool like the system modeller to manipulate Cedar systems. We believe the added complexity of the system modelling approach justifies the extra work required to build it and replace DF files. The success of DF files and the release process has demonstrated the need for automatic tools to manipulate versions and to integrate software that is developed by project members. The replacement of DF files by the Modeller will preserve the facilities provided by DF files and offer more complete facilities for automatic software development.

Although this thesis has dealt exclusively with Cedar and Mesa software management, the approaches described here can be applied to other systems. For example, the module interconnection structure of the C language is much simpler than that of Cedar, so the language

needed to describe a collection of C language files is correspondingly simpler than SM. There is a simple notion of dependency in C (an object file depends on one or more source files), however, and members of a project who want guarantees of completeness and consistency about versions of software could use DF files to describe their system. [Cristofor, *et al.*, 1980] contains a system similar to DF files that does not deal with problems of distributed computing.

The concept of a system modeller, or a tool to track changes to software, and to compile and load programs, is easily applied to other language systems. The Make program [Feldman, 1979] is a partial example of such a tool.

### 5.2 Future Research

System models are complete descriptions of Cedar systems. Their basic function will not change. Extensions of this research fall in the areas of 1) making better use of the information present in system models, and 2) making system models easier to use.

*Extensions into Databases*

Databases are naturally suited to storing modules and dependency relationships between modules. When the research in this thesis was started, there was no database system in Cedar that could handle the amount of data required. When such a database is available, I envision many programs that process data about modules in systems, such as sophisticated browsers and cross-reference tools.

An obvious and effective use of information in system models is as part of an editor or browser like Interlisp's MasterScope [Tietelman-Masinter, 1981] or PIE's Browser [Goldstein-Bobrow, 1980]. Such a tool would help the programmer by giving him information about the effect of his changes on other parts of the system. For example, models describe exactly where EXPORTed interfaces are used. Often the Cedar programmer wants to know whether procedures and variables in exported interfaces are used by any other modules. With a database of exporters and importers, a browser tool could easily display all clients of an interface. Similarly, a programmer who wants to see examples of other clients of an interface he is IMPORTing could be shown some by a browser.

Both examples are instances of the use of cross-reference information about a system that is too large to search through by hand. A system to maintain this kind of information was built using the Cedar database system [Brown, *et al.*, 1981] but was abandoned since the database was too large to be handled by the then-available transaction-based file server. A solution to this is being worked on as part of another research project at Xerox. Achieving acceptable performance from a centralized database in a distributed environment for Cedar system models remains a goal of database research at Xerox.

A database can also be used to store an evolutionary history of the system models in a project. The database could include "derivation" history about the kind of operation (a *Notice* or hand-editing) that produced a new version of a model.

By adoption of system models, we have an unambiguous description of Cedar software. Information about the system can be inserted by the Modeller Release Tool as it rewrites the models.

*Simplifying Common Use of Models*

Models written in the SML language appear to be very complicated because they express, in full detail, the relationships between Cedar modules. Models used in real-life are very large, although various defaulting rules are available that shorten the models considerably. Work will continue on the introduction of defaulting and other schemes to make the size of models smaller. We intend to make the most common use of the module interconnection facilities of Cedar as simple as possible. For example, in cases where there is only one version of each interface and only one EXPORTer of each interface, the list of interfaces and implementations could omit most information about interfaces since that can be derived by examining the implementation modules.

*Relaxation of Version Bindings*

The systems described here require that a reference to an object be either by an explicit version stamp (using a creation time or unique-id) or by no version, in which case, the newest version is used (">" in DF files, "!H" in models). Other systems (e.g., [Kaiser-Habermann, 1982] and [Horsley-Lynch, 1979]) allow more general version references such as "any version after June 12." There is a natural tension, however, between these types of "loose bindings" and the use of unique-ids. Loose bindings can hide important changes that affect programs, such as interface changes or changes to a package that alter its behavior. Loose bindings do not prevent new incompatible versions of software from being introduced into a system. Once the bindings are made "firm," there can be no question about the intended version.

*Integration with Source Code Control*

As noted in Section 2.1, the ability to control modification of modules by more than one programmer at a time was not a goal of this research. Systems like SCCS and the Pilot Librarian [Horsley-Lynch, 1979] control simultaneous updates to shared software files and are useful in other environments. These could be integrated into the systems presented in this thesis.

## References

[Avakian, *et al.*, 1982]
Arra Avakian, Sam Haradhvala, Julian Horn and Bruce Knobe, "The Design of an Integrated Support Software System," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 308-317, June 23-25, 1982.

[Brown, *et al.*, 1981]
Mark R. Brown, Roderic G. G. Cattell, and Norihisa Suzuki, "The Cedar DBMS: A Preliminary Report," *Proceedings of the SIGMOD Conference on Databases*, Ann Arbor, Michigan, May 1981.

[Cooprider, 1979]
Lee W. Cooprider, *The Representation of Families of Software Systems*, Ph.D Thesis, CMU Computer Science Department, CMU-CS-79-116, April 14, 1979.

[Cristofor, *et al.*, 1980]
Eugene Cristofor, T.A. Wendt, and B.C. Wonsiewicz, "Source Control + Tools = Stable Systems," *Proceedings of the Fourth Computer Software and Applications Conference*, pp. 527-532, October 29-31, 1980.

[Demers-Donahue, 1980]
A. Demers and J. Donahue, "Data Types, Parameters, and Type Checking," *Proceedings of the Seventh Symposium on Principles of Programming Languages*, Las Vegas, Nevada, pp. 12-23, 1980.

[DeRemer-Kron, 1976]
Frank DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, vol. 2, no. 2, pp. 80-86, June 1976.

[Deutsch-Taft, 1980]
L. Peter Deutsch and Edward A. Taft, "Requirements for an Experimental Programming Environment," Xerox PARC technical report CSL-80-10, June 1980.

[Dorado, 1981]
D. W. Clark, B. W. Lampson, G. A. McDaniel, S. M. Ornstein, and K. A. Pier, "The Dorado: A High Performance Personal Computer– Three Papers," Xerox PARC technical report CSL-81-1, January 1981.

[Feldman, 1979]
Stuart I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software Practice and Experience*, vol. 9 no. 4, April 1979.

[Glasser, 1978]
Alan L. Glasser, "The Evolution of a Source Code Control System," Proc. Software Quality and Assurance Workshop, *Software Engineering Notes*, vol. 3 no. 5, pp. 122-125, November 1978.

[Goldstein-Bobrow, 1980]
Ira P. Goldstein and Daniel G. Bobrow, "A Layered Approach to Software Design," Xerox PARC technical report CSL-80-5, December 1980.

[Goldstein-Bobrow, 1980]
Ira P. Goldstein and Daniel G. Bobrow, "Descriptions for a Programming Environment," *Proceedings of the First Annual Conference of the National Association of Artificial Intelligence*, Stanford, California, August 1980.

[Goldstein-Bobrow, 1980]
Ira P. Goldstein and Daniel G. Bobrow, "Representing Design Alternatives," *Proceedings of the Artificial Intelligence and Simulation of Behavior Conference*, Amsterdam, July 1980.

[Habermann, 1979a]
A. Nico Habermann, "Tools for Software System Construction," *Proceedings of the Software Tools Workshop*, Boulder, Colorado, May 1979.

[Habermann, *et al.*, 1982]
A. Nico Habermann, Robert Ellison, Raul Medina-Mora, Peter Feiler, David S. Notkin, Gail E. Kaiser, David B. Garlan and Steven Popovich, "The Second Compendium of Gandalf Documentation," CMU Department of Computer Science, May 24, 1982.

[Habermann-Perry, 1980]
A. Nico Habermann and Dewayne E. Perry, "System Compositions and Version Control for Ada," CMU Computer Science Department, May 1980.

[Harslem-Nelson, 1982]
Eric Harslem and LeRoy E. Nelson, "A Retrospective on the Development of Star," *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982.

[Horsley-Lynch, 1979]
Thomas R. Horsley and William C. Lynch, "Pilot: A Software Engineering Case Study," *Proceedings of the 4th International Conference on Software Engineering*, pp. 94-99, 1979.

[Ivie, 1977]
Evan L. Ivie, "The Programmer's Workbench- A Machine for Software Development," *Communications of the ACM*, vol. 20 no. 10, pp. 746-753, October 1977.

[Joy, 1981]
William N. Joy, Measurements of 4.1 BSD, personal communication, May 1981.

[Kaiser-Habermann, 1982]
Gail E. Kaiser and A. Nico Habermann, "An Environment for System Version Control," in "The Second Compendium of Gandalf Documentation," CMU Department of Computer Science, February 4, 1982.

[Kernighan-Ritchie, 1978]
Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Lauer-Satterthwaite, 1979]
Hugh C. Lauer and Edwin H. Satterthwaite, "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering*, pp. 174-182, 1979.

[Mitchell, *et al.*, 1979]
James G. Mitchell, William Maybury, and Richard Sweet, "Mesa Language Manual, version 5.0," Xerox PARC technical report CSL-79-3, April 1979.

[Morris, 1982]
James H. Morris, personal communication, 1982.

[Nelson, 1981]
Bruce J. Nelson, "Remote Procedure Call," Xerox PARC technical report CSL-81-9, May 1981.

[Redell, *et al.*, 1979]
D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell, "Pilot: an Operating System for a Personal Computer," *Proceedings of the Seventh Symposium on Operating System Principles*, December 1979.

[Rochkind, 1975]
Marc J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, Vol. 1, No. 4, pp. 364-370, December 1975.

[Teitelman-Masinter, 1981]
Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," *Computer*, vol. 14 no. 4, pp. 25-34, April 1981.

[Thacker, *et al.*, 1982]
C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull and D. R. Boggs, "Alto: A Personal Computer" in *Computer Structure: Principles and Examples*, D. Siewiorek, D. G. Bell and A. Newell, editors, McGraw-Hill, 1982.

[Tichy, 1980]
Walter F. Tichy, *Software Development Control Based on System Structure Description*, Ph.D. Thesis, CMU Computer Science Department, CMU-CS-80-120, January 1980.

[Tichy, 1982]
W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982.

# Appendix A: Statistics about Cedar Releases

Figure A.1 is a table of all automated Cedar releases. It shows the name of the release, the date the release was announced (which is usually the day phase three was run), and has statistics about components in each release. A component is a DF file or collection of DF files that are thought of as a package. The announcement of a new release includes an entry for each component with information about any changes. Notable in this figure is the 3.2 release, which took 8 weeks of development, twice the normal time, due to some mistakes and summer vacations.

| Release | Date | Time after previous | # Components | # New | # Changed | # Withdrawn |
|---|---|---|---|---|---|---|
| 2.0 | October 10, 1981 | -- | 22 | 22 | 0 | 0 |
| 2.1 | November 8 | 4 weeks | 24 | 2 | 9 | 0 |
| 2.2 | December 17 | 5 weeks | 26 | 2 | 14 | 0 |
| 2.3 | January 22 | 5 weeks | 32 | 6 | 6 | 0 |
| 2.4 | February 24 | 4 weeks 2 days | 40 | 8 | 25 | 1 |
| 2.4.1 | February 26 | 2 days | 40 | 0 | 2 | 0 |
| 2.5 | March 15 | 3 weeks | 48 | 8 | 26 | 0 |
| 2.5.1 | March 19 | 4 days | 48 | 0 | 2 | 0 |
| 2.6 | April 1 | 2 weeks | 50 | 2 | 17 | 1 |
| 3.0 | May 7 | 5 weeks | 62 | 18 | 36 | 4 |
| 3.1 | May 21 | 2 weeks | 63 | 1 | 4 | 0 |
| 3.2 | July 23 | 8 weeks | 79 | 17 | 61 | 4 |
| 3.3 | August 20 | 4 weeks | 78 | 0 | 78 | 1 |
| 3.4 | October 7 | 7 weeks | 87 | 9 | 78 | 0 |
| 3.4.1 | October 14 | 1 week | 87 | 0 | 5 | 0 |
| 3.5 | December 1 | 6 weeks 3 days | 95 | 11 | 65 | 3 |
| 3.5.1 | December 14 | 2 weeks | 96 | 1 | 18 | 1 |
| 3.5.2 | December 17 | 3 days | 96 | 0 | 2 | 0 |

Release Summary

Figure A.1

Figure A.2 shows the corresponding table for the first internal release made by the Mesa Group of the Systems Development Division of Xerox.

| Release | Date | Time after previous | # Components | # New | # Changed | # Withdrawn |
|---|---|---|---|---|---|---|
| 1.0 | September 5, 1982 | -- | 89 | 89 | 0 | 0 |

Release Summary

Figure A.2

Figure A.3 is a table describing the release from the standpoint of the amount of work done by the Release Tool. Since some files do not change between releases, the Release Tool only stores those that do not change. The number of files and megabytes actually stored are smaller than the number of files and megabytes for the entire release. The number of DF files stored refers to the DF files rewritten by phase three of the Release Tool. The running times columns for phases one through three include a number of estimates because of a number of technical problems. Notable is the 6 hours required to copy the 3.2 release, due to the large number of file changes and an overloaded file server. This table shows that, if the two emergency releases are excluded, Cedar has normally followed a major release with many changes, by a minor release with considerably fewer changes.

| Release | # files actually stored | # bytes actually copied | # DF files stored | Running Times (Hours:Mins) | | |
|---|---|---|---|---|---|---|
| | | | | Phase 1 | Phase 2 | Phase 3 |
| 2.0 | 610 | 12mb | 34 | :27 | ? | :58 |
| 2.1 | 303 | 7.8mb | 21 | :34 | ? | :53 |
| 2.2 | 808 | 15.4mb | 57 | :50 | :49 | 1:15 |
| 2.3 | 200 | 5.1mb | 30 | :54 | ? | 1:0 |
| 2.4 | 1747 | 32mb | 72 | ? | :46 | 2:03 |
| 2.4.1 | 64 | 5.5mb | 8 | :20 | -- | :12 |
| 2.5 | 1411 | 25.1mb | 67 | ? | :50 | 1:57 |
| 2.5.1 | 18 | 3.9mb | 8 | :03 | -- | :07 |
| 2.6 | 498 | 16.2mb | 62 | ? | :30 | 1:07 |
| 3.0 | ? (large) | ? (large) | 102 | 1:15 | 1:11 | 3:0 |
| 3.1 | ? (small) | 11.6mb | 25 | :20 | ? | 1:0 |
| 3.2 | 3217 | 59.3mb | 138 | 1:30 | 1:0 | 6:0 |
| 3.3 | 2477 | 42.5mb | 144 | 1:15 | 1:0 | 3:30 |
| 3.4 | 3229 | 60.0mb | 163 | 1:01 | 1:30 | 6:15 |
| 3.4.1 | 179 | 10.5mb | 30 | :15 | :30 | :40 |
| 3.5 | 1871 | 20489 pgs | 120 | :44 | 1:20 | 2:58 |
| 3.5.1 | 358 | 9270pgs | 58 | :28 | 1:03 | 1:11 |
| 3.5.2 | 13 | 2796 | 13 | :04 | -- | :11 |

Differential Release File Movement

Figure A.3

Figure A.4 shows the corresponding tables for the release of the Mesa group. Because of machine differences, the times to make releases are not comparable.

| Release | # files actually stored | # bytes actually copied | # DF files stored | Running Times (Hours:Mins) | | |
|---|---|---|---|---|---|---|
| | | | | Phase 1 | Phase 2 | Phase 3 |
| 1.0 | 2578 | 40.8mb | 127 | 1:57 | 2:0 | 4:30 |

Differential Release File Movement

Figure A.4

Figure A.5 is a table of statistics on the release from the standpoint of a user. Some of the files, etc., counted in one release are also counted in subsequent releases. Releases before 2.6 were not measured for this table. The table includes subsections for object and source files. In the earlier releases there are more object files, in the later releases there are more source files. The surplus of object files is due to the use of files that are outside the release envelope, which were eliminated in later releases. The surplus of some files in later releases is due to test programs whose sources are saved but whose object files are not needed. Also, note the size of Cedar decreased between 3.2 and 3.3 because a large program (a debugger) was eliminated from the release.

| Release | Object Files | | | | | Source Files | | | | | | Total | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Files | Defs | Impls | Configs | Mbytes | Files | Defs | Impls | Configs | Lines | Mbytes | Files | Mbytes | DF files |
| 2.6 | 1724 | 776 | 832 | 116 | 24.4 | 1527 | 655 | 765 | 107 | 320,354 | 11.5 | 3674 | 44.6 | 129 |
| 3.0 | 1981 | 903 | 942 | 138 | 26.5 | 1801 | 779 | 892 | 130 | 373,477 | 13.5 | 4219 | 50.3 | 144 |
| 3.1 | 2000 | 909 | 954 | 137 | 26.7 | 1817 | 783 | 903 | 131 | 377,081 | 13.6 | 4247 | 50.7 | 145 |
| 3.2 | 2098 | 923 | 1033 | 142 | 26.8 | 2029 | 857 | 1026 | 146 | 432,284 | 15.9 | 4670 | 56.9 | 153 |
| 3.3 | 1983 | 857 | 994 | 132 | 24.2 | 2005 | 856 | 1010 | 139 | 424,248 | 15.7 | 4511 | 53.7 | 153 |
| 3.4 | 2051 | 865 | 1050 | 136 | 26.6 | 2084 | 874 | 1067 | 143 | 447,310 | 17.2 | 4731 | 61.4 | 170 |
| 3.4.1 | 2051 | 865 | 1049 | 137 | 26.6 | 2083 | 874 | 1066 | 143 | 447,270 | 17.2 | 4730 | 61.5 | 170 |
| 3.5 | 2100 | 888 | 1072 | 140 | 25.1 | 2133 | 897 | 1091 | 145 | 457,259 | 17.9 | 4843 | 63.5 | 162 |
| 3.5.1 | 2107 | 893 | 1073 | 141 | 27.5 | 2139 | 901 | 1092 | 146 | 457,248 | 18.0 | 4859 | 63.7 | 162 |
| 3.5.2 | 2107 | 893 | 1073 | 141 | 27.5 | 2139 | 901 | 1092 | 146 | 457,274 | 18.0 | 4863 | 63.7 | 162 |

Complete Release Totals

Figure A.5

Figure A.6 is a table that shows the corresponding table for the first release of the Mesa group.

| Release | Object Files | | | | | Source Files | | | | | | Total | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Files | Defs | Impls | Configs | Mbytes | Files | Defs | Impls | Configs | Lines | Mbytes | Files | Mbytes | DF files |
| 1.0 | 1542 | | | | 17.9 | 1495 | 624 | 754 | 117 | 325,683 | 11.7 | 3490 | 41.6 | 127 |

Complete Release Totals

Figure A.6

# Appendix B: Example DF Files

The DF file below was submitted to the Cedar 3.4 release. It describes a component named IO that is implemented by a configuration named IOPackage. The public interfaces are named in the second section named "Exports." The implementation consists of a number of modules that end in "Impl." IO imports many other components after the list of implementors.

Notations like {n} refer to numbered points in the "Notes" section below.

```
// IO.df
// Last Modified On September 17, 1982 3:01 pm By Warren Teitelman   {9}

Exports [Indigo]<PreCedar>Top>     ReleaseAs [Indigo]<Cedar>Top>   {1}

  IO.Df                                  6-Oct-82 21:07:47 PDT

-- Interfaces
Exports [Indigo]<PreCedar>IO>     ReleaseAs [Indigo]<Cedar>IO> {2}

  IO.mesa!6 {10}                         8-Sep-82 22:26:35 PDT
  ReadMacros.mesa!1                      9-Jun-82 21:07:57 PDT
  FileIO.mesa!3                          1-Sep-82 15:42:46 PDT
  XPrivateIO.Mesa!3                     21-Sep-82 21:05:54 PDT
  FileIOPrivate.mesa!2                  30-Aug-82 14:03:19 PDT
  PrintTV.mesa!1                         1-Sep-82 20:33:38 PDT
  Juniper.mesa!2                        26-Aug-82 15:09:50 PDT
  UserProfile.mesa!4                     1-Sep-82 22:27:26 PDT

  IO.bcd!7                               8-Sep-82 23:41:58 PDT
  ReadMacros.bcd!6                       8-Sep-82 23:42:11 PDT
  FileIO.bcd!6                           8-Sep-82 23:42:21 PDT
  XPrivateIO.bcd!6                      21-Sep-82 21:06:10 PDT
  FileIOPrivate.bcd!6                    8-Sep-82 23:42:24 PDT
  PrintTV.bcd!1                          1-Sep-82 20:33:58 PDT
  Juniper.bcd!6                          8-Sep-82 23:42:15 PDT
  UserProfile.bcd!6                      8-Sep-82 23:42:17 PDT

  IO.comments!2                          1-Oct-82 13:08:00 PDT
  PF.comments!1                         23-Jun-82 13:06:35 PDT
  io.changes!9                          29-Sep-82 12:51:23 PDT

-- Documentation
Directory [Indigo]<PreCedar>IO>     ReleaseAs [Indigo]<Cedar>Documentation>     {6}

  IO.press!2                            17-Sep-82 14:48:36 PDT
  FileIO.press!1                        17-Sep-82 14:47:17 PDT

-- implementation
Directory [Indigo]<PreCedar>IO>     ReleaseAs [Indigo]<Cedar>IO>  {4}

  +IOPackage.bcd!34                      6-Oct-82 20:18:00 PDT {3}
  IOPackage.config!7                    28-Sep-82 14:09:30 PDT

  IOImpl.mesa!3                          1-Oct-82 13:12:26 PDT
  InputImpl.mesa!1                       5-Aug-82 13:48:40 PDT
  OutputImpl.mesa!8                     21-Sep-82 21:05:32 PDT
  SomeStreamsImpl.mesa!4                 4-Oct-82 21:01:02 PDT
  IOPFImpl.mesa!3                        8-Sep-82 23:54:04 PDT
  TVStreamImpl.mesa!7                   29-Sep-82 15:44:23 PDT
  PrintTVImpl.mesa!17                    6-Oct-82 20:15:27 PDT
  PrintTypeImpl.mesa!7                  23-Sep-82 13:30:40 PDT
```

```
  FileIOCommonImpl.mesa!3                 1-Sep-82 21:37:49 PDT
  FileIOJuniperImpl.mesa!4               22-Sep-82 17:24:30 PDT
  FileIOPilotImpl.mesa!6                  5-Oct-82 13:20:17 PDT
  UserProfileImpl.mesa!7                 27-Sep-82 16:10:15 PDT
  CoPilotIO.mesa!2                        1-Sep-82 22:35:18 PDT

  IOImpl.bcd!8                            5-Oct-82 11:51:29 PDT
  InputImpl.bcd!7                         5-Oct-82 11:52:08 PDT
  OutputImpl.bcd!8                       21-Sep-82 21:07:07 PDT
  SomeStreamsImpl.bcd!8                   5-Oct-82 11:52:42 PDT
  IOPFImpl.bcd!6                          5-Oct-82 11:54:23 PDT
  TVStreamImpl.bcd!7                     29-Sep-82 15:44:58 PDT
  PrintTVImpl.bcd!19                      6-Oct-82 20:16:24 PDT
  PrintTypeImpl.bcd!9                    23-Sep-82 14:08:35 PDT
  FileIOCommonImpl.bcd!5                 10-Sep-82 13:25:51 PDT
  FileIOJuniperImpl.bcd!5                22-Sep-82 17:24:58 PDT
  FileIOPilotImpl.bcd!8                   5-Oct-82 13:22:25 PDT
  UserProfileImpl.bcd!8                  27-Sep-82 16:14:02 PDT
  CoPilotIO.bcd!6                        10-Sep-82 13:26:48 PDT

  compileio.cm!6                          8-Sep-82 23:34:19 PDT {5}

-- test program (maybe belong in another df file)

  FileIOTestImpl.mesa!1                  25-Jun-82 15:29:55 PDT
  FileIOTest.config!1                    25-Jun-82 15:34:40 PDT

Imports [Indigo]<PreCedar>Top>ListsAndAtoms.Df Of >
    Using [Atom.bcd, List.bcd] {7}, {8}

Imports [Indigo]<PreCedar>Top>Rigging.Df Of >
    Using [Convert.bcd, Convertunsafe.bcd, RefText.bcd, Rope.bcd, RopeInline.bcd]

Imports [Indigo]<PreCedar>Top>Runtime.df Of >
    Using [RTBasic.bcd, RTTypesBasic.bcd, RTTBridge.bcd, RTTRemoteBridge.bcd,
       RTTypes.bcd, RTTypesExtras.bcd, SafeStorage.bcd]

Imports [Indigo]<PreCedar>TeleDebug>WorldVM.df Of >
    Using [WorldVM.bcd]

Imports [Indigo]<PreCedar>Top>CedarReals.df Of >
    Using [Real.bcd, RealOps.bcd, Ieee.bcd]

Imports [Indigo]<PreCedar>Top>PilotInterfaces.df Of 20-Aug-82 15:29:55 PDT
    Using [ByteBlt.bcd, Environment.bcd, File.bcd, Heap.bcd, Inline.bcd,
       PageFault.bcd, Pilotswitches.bcd, PrincOps.bcd, Process.bcd,
       Runtime.bcd, RuntimeInternal.bcd, Space.bcd, System.bcd, TemporaryBooting.bcd,
       Transaction.bcd, UserTerminal.bcd, WriteFault.bcd]

Exports Imports [Indigo]<PreCedar>Top>CompatibilityPackage.df Of >
    Using [Ascii.bcd, DCSFileTypes.bcd, Directory.bcd, FileStream.bcd,
       PropertyTypes.bcd, String.bcd, Time.bcd]

Exports Imports [Indigo]<PreCedar>Top>UserCredentials.df Of >
    Using [UserCredentials.bcd]

Imports [Indigo]<PreCedar>Top>Pine.df Of >
    Using [CommonPineDefs.bcd, UserPineDefs.bcd]

Imports [Indigo]<PreCedar>Top>CIFS.df Of >
    Using [CIFS.bcd]

Imports [Indigo]<PreCedar>Top>CedarSnapshot.df Of >
    Using [CedarSnapshot.bcd]
```

*Notes*

1) The DF file has a self-reference. Because of it, IO.DF will be released to a special directory <Cedar>Top>.

2) The interface files are put on <Cedar>IO>. "Exports" gives an indication that clients may want to Import these files.

3) An object file is preceded by a "+", indicating to VerifyDF that this is a root of the dependency graph.

4) The implementation files are put on <Cedar>IO>.

5) We encourage implementors add to their DF files command files that compile and bind their software.

6) Documentation is released onto <Cedar>Documentation>.

7) Interfaces needed to compile the sources are Imported from appropriate DF files.

8) Interfaces are Imported from a DF file that is on a working directory, referring to the newest version.

9) The "--" at the beginning of a line signifies a comment.

10) Filenames like "IO.Mesa!6" refer to file named "IO.Mesa", version number 6 on the file server. PARC's file servers can store multiple versions of a file on a directory, which are numbered starting at 1. The version number "!6" is entirely optional, since the create time identifies the version of "IO.Mesa" desired. If present, a version number is used as a hint.

*After the Release*

The DF file is rewritten by Phase Three of the Release Tool as follows.

```
// IO.df
// Last Modified On September 17, 1982 3:01 pm By Warren Teitelman


Exports [Indigo]<Cedar>Top>     CameFrom [Indigo]<PreCedar>Top>

  IO.Df                                    7-Oct-82 11:51:45 PDT

-- Interfaces
Exports [Indigo]<Cedar>IO>     CameFrom [Indigo]<PreCedar>IO>

  IO.mesa!2                                8-Sep-82 22:26:35 PDT
  ReadMacros.mesa!1                        9-Jun-82 21:07:57 PDT
  FileIO.mesa!2                            1-Sep-82 15:42:46 PDT
  XPrivateIO.Mesa!1                       21-Sep-82 21:05:54 PDT
  FileIOPrivate.mesa!2                    30-Aug-82 14:03:19 PDT
```

```
  PrintTV.mesa!1                       1-Sep-82 20:33:38 PDT
  Juniper.mesa!2                      26-Aug-82 15:09:50 PDT
  UserProfile.mesa!2                   1-Sep-82 22:27:26 PDT

  IO.bcd!3                             8-Sep-82 23:41:58 PDT
  ReadMacros.bcd!3                     8-Sep-82 23:42:11 PDT
  FileIO.bcd!3                         8-Sep-82 23:42:21 PDT
  XPrivateIO.bcd!1                    21-Sep-82 21:06:10 PDT
  FileIOPrivate.bcd!3                  8-Sep-82 23:42:24 PDT
  PrintTV.bcd!1                        1-Sep-82 20:33:58 PDT
  Juniper.bcd!3                        8-Sep-82 23:42:15 PDT
  UserProfile.bcd!3                    8-Sep-82 23:42:17 PDT

  IO.comments!2                        1-Oct-82 13:08:00 PDT
  PF.comments!1                       23-Jun-82 13:06:35 PDT
  io.changes!2                        29-Sep-82 12:51:23 PDT

-- Documentation
Directory [Indigo]<Cedar>Documentation>   CameFrom [Indigo]<PreCedar>IO>

  IO.press!1                          17-Sep-82 14:48:36 PDT
  FileIO.press!1                      17-Sep-82 14:47:17 PDT

-- implementation
Directory [Indigo]<Cedar>IO>    CameFrom [Indigo]<PreCedar>IO>

 +IOPackage.bcd!3                      6-Oct-82 20:18:00 PDT
  IOPackage.config!2                  28-Sep-82 14:09:30 PDT

  IOImpl.mesa!2                        1-Oct-82 13:12:26 PDT
  InputImpl.mesa!2                     5-Aug-82 13:48:40 PDT
  OutputImpl.mesa!2                   21-Sep-82 21:05:32 PDT
  SomeStreamsImpl.mesa!2               4-Oct-82 21:01:02 PDT
  IOPFImpl.mesa!2                      8-Sep-82 23:54:04 PDT
  TVStreamImpl.mesa!2                 29-Sep-82 15:44:23 PDT
  PrintTVImpl.mesa!1                   6-Oct-82 20:15:27 PDT
  PrintTypeImpl.mesa!1                23-Sep-82 13:30:40 PDT
  FileIOCommonImpl.mesa!2              1-Sep-82 21:37:49 PDT
  FileIOJuniperImpl.mesa!2            22-Sep-82 17:24:30 PDT
  FileIOPilotImpl.mesa!2               5-Oct-82 13:20:17 PDT
  UserProfileImpl.mesa!2              27-Sep-82 16:10:15 PDT
  CoPilotIO.mesa!1                     1-Sep-82 22:35:18 PDT

  IOImpl.bcd!3                         5-Oct-82 11:51:29 PDT
  InputImpl.bcd!3                      5-Oct-82 11:52:08 PDT
  OutputImpl.bcd!3                    21-Sep-82 21:07:07 PDT
  SomeStreamsImpl.bcd!3                5-Oct-82 11:52:42 PDT
  IOPFImpl.bcd!3                       5-Oct-82 11:54:23 PDT
  TVStreamImpl.bcd!3                  29-Sep-82 15:44:58 PDT
  PrintTVImpl.bcd!1                    6-Oct-82 20:16:24 PDT
  PrintTypeImpl.bcd!1                 23-Sep-82 14:08:35 PDT
  FileIOCommonImpl.bcd!3              10-Sep-82 13:25:51 PDT
  FileIOJuniperImpl.bcd!3             22-Sep-82 17:24:58 PDT
  FileIOPilotImpl.bcd!3                5-Oct-82 13:22:25 PDT
  UserProfileImpl.bcd!3               27-Sep-82 16:14:02 PDT
  CoPilotIO.bcd!1                     10-Sep-82 13:26:48 PDT

  compileio.cm!2                       8-Sep-82 23:34:19 PDT

-- test program (maybe belong in another df file)

  FileIOTestImpl.mesa!1               25-Jun-82 15:29:55 PDT
  FileIOTest.config!1                 25-Jun-82 15:34:40 PDT

Imports [Indigo]<Cedar>Top>ListsAndAtoms.Df  Of  7-Oct-82 11:51:50 PDT
```

```
    CameFrom [Indigo]<PreCedar>Top>
    Using [Atom.bcd, List.bcd]

Imports [Indigo]<Cedar>Top>Rigging.Df  Of  7-Oct-82 11:52:30 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [Convert.bcd, Convertunsafe.bcd, RefText.bcd, Rope.bcd, RopeInline.bcd]

Imports [Indigo]<Cedar>Top>Runtime.df  Of  7-Oct-82 11:52:35 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [RTBasic.bcd, RTTypesBasic.bcd, RTTBridge.bcd, RTTRemoteBridge.bcd,
       RTTypes.bcd, RTTypesExtras.bcd, SafeStorage.bcd]

Imports [Indigo]<Cedar>TeleDebug>WorldVM.df  Of  7-Oct-82 11:53:22 PDT
    CameFrom [Indigo]<PreCedar>TeleDebug>
    Using [WorldVM.bcd]

Imports [Indigo]<Cedar>Top>CedarReals.df  Of  7-Oct-82 11:51:04 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [Real.bcd, RealOps.bcd, Ieee.bcd]

Imports [Indigo]<Cedar>Top>PilotInterfaces.df  Of  20-Aug-82 15:29:55 PDT
    Using [ByteBlt.bcd, Environment.bcd, File.bcd, Heap.bcd, Inline.bcd,
       PageFault.bcd, Pilotswitches.bcd, PrincOps.bcd, Process.bcd,
       Runtime.bcd, RuntimeInternal.bcd, Space.bcd, System.bcd, TemporaryBooting.bcd,
       Transaction.bcd, UserTerminal.bcd, WriteFault.bcd]

Exports Imports [Indigo]<Cedar>Top>CompatibilityPackage.df  Of  7-Oct-82 11:51:15 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [Ascii.bcd, DCSFileTypes.bcd, Directory.bcd, FileStream.bcd,
       PropertyTypes.bcd, String.bcd, Time.bcd]

Exports Imports [Indigo]<Cedar>Top>UserCredentials.df  Of  7-Oct-82 11:53:04 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [UserCredentials.bcd]

Imports [Indigo]<Cedar>Top>Pine.df  Of  7-Oct-82 11:52:17 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [CommonPineDefs.bcd, UserPineDefs.bcd]

Imports [Indigo]<Cedar>Top>CIFS.df  Of  7-Oct-82 11:51:07 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [CIFS.bcd]

Imports [Indigo]<Cedar>Top>CedarSnapshot.df  Of  7-Oct-82 11:51:05 PDT
    CameFrom [Indigo]<PreCedar>Top>
    Using [CedarSnapshot.bcd]
```

Notice the "ReleaseAs" phrases have been replaced with "CameFrom" phrases for most of the entries, and the ">" dates have been replaced by the date of the new version of each DF file on <Cedar>Top>.

The following pages contain a copy of `Runtime.DF` that was submitted to the Cedar 3.4 release. `Runtime.DF` is one of the larger DF files in the release and is shown in Figure 2.8 at the end of Chapter 2, in a dependency cycle with IO.DF. It is included here to show the size of file that is handled by the DF system.

```
-- Runtime.df
-- A model file for the Cedar runtime system
-- Last Modified On October 4, 1982 1:59 pm By Paul Rovner
```

```
Exports [Indigo]<PreCedar>Top>     ReleaseAs [Indigo]<Cedar>Top>

  Runtime.df                               4-Oct-82 11:02:46 PDT

Exports [Indigo]<PreCedar>Runtime>     ReleaseAs [Indigo]<Cedar>Runtime>

  AtomsPrivate.bcd!1                       1-Sep-82 17:32:54 PDT
  AtomsPrivate.mesa!1                     20-May-82 10:34:46 PDT
  RTBasic.bcd!1                            5-Aug-82 14:14:07 PDT
  RTBasic.mesa!1                          25-May-82 14:30:13 PDT
  RTLoader.bcd!1                           1-Sep-82 17:32:53 PDT
  RTLoader.mesa!1                         19-May-82 19:18:06 PDT
  RTMiniModel.bcd!1                        1-Sep-82 17:34:20 PDT
  RTMiniModel.mesa!1                      27-May-82 12:51:09 PDT
  RTModel.bcd!1                            4-Oct-82 12:09:04 PDT
  RTModel.mesa!1                           4-Oct-82 11:49:56 PDT
  RTOS.bcd!1                               9-Sep-82 10:33:24 PDT
  RTOS.mesa!1                              9-Sep-82 10:32:09 PDT
  RTProcess.bcd!1                          9-Sep-82 10:33:26 PDT
  RTProcess.mesa!1                         9-Sep-82  9:34:32 PDT
  RTRefCounts.bcd!1                        1-Sep-82 17:33:28 PDT
  RTRefCounts.mesa!1                      20-May-82 10:46:59 PDT
  RTStart.bcd!1                            6-Aug-82  9:59:55 PDT
  RTStart.mesa!1                           4-Jun-81 11:06:10 PDT
  RTStorageOps.bcd!1                       1-Sep-82 17:32:58 PDT
  RTStorageOps.mesa!1                     20-May-82 13:56:33 PDT
  RTTypesBasic.bcd!1                       5-Aug-82 14:55:10 PDT
  RTTypesBasic.mesa!1                     27-May-82 12:51:05 PDT
  RTTypesBasicPrivate.bcd!1                1-Sep-82 17:33:13 PDT
  RTTypesBasicPrivate.mesa!1              25-May-82 15:24:41 PDT
  RTZones.bcd!1                            1-Sep-82 17:33:21 PDT
  RTZones.mesa!1                          20-May-82 10:43:53 PDT
  SafeStorage.bcd!1                        5-Aug-82 14:14:09 PDT
  SafeStorage.mesa!1                      27-May-82 12:51:36 PDT
  UnsafeStorage.bcd!1                      6-Aug-82 10:00:04 PDT
  UnsafeStorage.mesa!1                    20-May-82 10:06:51 PDT
Directory [Indigo]<PreCedar>Runtime>     ReleaseAs [Indigo]<Cedar>Documentation>

  RTTypes.tioga!1                          3-Aug-82 11:24:48 PDT
  RTTypes.press!1                          3-Aug-82 11:38:07 PDT
  Runtime.tioga!1                          3-Aug-82 11:27:54 PDT
  Runtime.press!1                          3-Aug-82 11:28:54 PDT

Directory [Indigo]<PreCedar>Runtime>     ReleaseAs [Indigo]<Cedar>Runtime>

  +RT.bcd!1                                23-Sep-82 10:59:19 PDT
  RT.config!1                             23-Sep-82 10:44:50 PDT
  Processes.mesa!1                        26-Aug-82 11:24:15 PDT
  CompileRuntime.cm!1                      4-Oct-82 11:59:42 PDT
  I-RT.cm!1                                4-Oct-82 12:00:15 PDT
  L-RT.cm!1                                4-Oct-82 12:01:57 PDT
  MC-RT.cm!1                               4-Oct-82 12:01:19 PDT
  CountPinnedPages.mesa!1                 30-Apr-81 19:00:22 PDT
  CreateCedarVM.mesa!1                     2-Dec-81  1:14:39 PST
  PrintInUseFileNames.mesa!1              21-Jun-82 14:57:11 PDT
  PTestQ.bcd!1                            28-Sep-82 16:21:53 PDT
  PTestQ.mesa!1                           27-Sep-82 15:46:44 PDT
  Test.config!1                           27-Sep-82 14:34:52 PDT
  +Test.bcd!1                              28-Sep-82 16:56:40 PDT
  ttt.mesa!1                               2-Oct-82 16:23:18 PDT
  ttt.bcd!1                                4-Oct-82 12:30:42 PDT
  GCTableFaultRecorder.mesa!1              6-Aug-82 10:20:11 PDT
  GCTableFaultRecorder.bcd!1               6-Aug-82 10:20:39 PDT
  RTBases.bcd!1                            1-Sep-82 17:32:57 PDT
  RTBases.mesa!1                          20-May-82 10:10:27 PDT
```

```
RTCommon.bcd!1                          1-Sep-82 17:33:05 PDT
RTCommon.mesa!1                        20-May-82 10:15:20 PDT
RTFlags.bcd!1                           6-Aug-82 10:00:02 PDT
RTFlags.mesa!1                          5-Feb-82 10:46:08 PST
RTMicrocode.bcd!1                       1-Sep-82 17:33:34 PDT
RTMicrocode.mesa!1                     20-May-82 10:18:43 PDT
RTProcessPrivate.bcd!1                  9-Sep-82 10:33:30 PDT
RTProcessPrivate.mesa!1                 9-Sep-82 10:33:08 PDT
RTQuanta.bcd!1                          1-Sep-82 17:32:43 PDT
RTQuanta.mesa!1                        20-May-82 10:22:11 PDT
RTQueue.bcd!1                           6-Aug-82 10:00:01 PDT
RTQueue.mesa!1                          3-Jun-81 15:19:10 PDT
RTSponge.bcd!1                          1-Sep-82 17:32:36 PDT
RTSponge.mesa!1                         5-Mar-82  9:23:36 PST
RTStorageAccounting.bcd!1              23-Sep-82 10:43:01 PDT
RTStorageAccounting.mesa!1             23-Sep-82 10:38:02 PDT
Runs.bcd!1                              6-Aug-82  9:59:56 PDT
Runs.mesa!1                            27-Mar-81 11:01:36 PST
AtomsPrivateImpl.bcd!1                  7-Sep-82 14:13:44 PDT
AtomsPrivateImpl.mesa!1                 8-Mar-82 10:35:47 PST
RCMapWalkerImpl.bcd!1                   7-Sep-82 14:14:09 PDT
RCMapWalkerImpl.mesa!1                  8-Mar-82 10:51:20 PST
RTAllocatorImpl.bcd!1                  23-Sep-82 10:43:28 PDT
RTAllocatorImpl.mesa!1                 23-Sep-82 10:40:09 PDT
RTBasesImpl.bcd!1                       9-Sep-82 10:34:29 PDT
RTBasesImpl.mesa!1                     27-May-82 16:11:40 PDT
RTLoaderImpl.bcd!1                      9-Sep-82 10:35:12 PDT
RTLoaderImpl.mesa!1                    20-May-82 10:54:44 PDT
RTOSImpl.bcd!1                          9-Sep-82 10:35:29 PDT
RTOSImpl.mesa!1                         9-Sep-82  8:31:18 PDT
RTPageFaultImpl.bcd!1                   9-Sep-82 10:35:39 PDT
RTPageFaultImpl.mesa!1                 16-Mar-82 11:09:07 PST
RTPrefAllocImpl.bcd!1                  23-Sep-82 10:44:58 PDT
RTPrefAllocImpl.mesa!1                 23-Sep-82 10:39:25 PDT
RTProcessImpl.bcd!1                     9-Sep-82 10:35:56 PDT
RTProcessImpl.mesa!1                    9-Sep-82  9:34:45 PDT
RTProcessPrivateImpl.bcd!1              9-Sep-82 10:40:32 PDT
RTProcessPrivateImpl.mesa!1             9-Sep-82 10:38:55 PDT
RTQueueImpl.bcd!1                       6-Aug-82 10:13:36 PDT
RTQueueImpl.mesa!1                      3-Sep-81 13:23:41 PDT
RTReclaimerImpl.bcd!1                   9-Sep-82 10:36:13 PDT
RTReclaimerImpl.mesa!1                 20-May-82 13:17:02 PDT
RTRefAccounting.bcd!1                   7-Sep-82 14:16:44 PDT
RTRefAccounting.mesa!1                  8-Feb-82 11:47:57 PST
RTRefCountsImpl.bcd!1                  23-Sep-82 10:45:22 PDT
RTRefCountsImpl.mesa!1                  7-Sep-82 16:31:13 PDT
RTSpongeImpl.bcd!1                      9-Sep-82 10:36:45 PDT
RTSpongeImpl.mesa!1                     5-Mar-82  9:23:53 PST
RTStartImpl.bcd!1                       9-Sep-82 10:42:56 PDT
RTStartImpl.mesa!1                      9-Sep-82 10:42:27 PDT
RTStopProcessImpl.bcd!1                 9-Sep-82 10:36:54 PDT
RTStopProcessImpl.mesa!1                9-Sep-82  7:49:15 PDT
RTStorageAccountingImpl.bcd!1          23-Sep-82 10:45:39 PDT
RTStorageAccountingImpl.mesa!1         27-May-82 16:18:03 PDT
RTSymbolAccessImpl.bcd!1                9-Sep-82 10:37:02 PDT
RTSymbolAccessImpl.mesa!1               6-Aug-82 10:15:03 PDT
RTTraceAndSweepImpl.bcd!1               9-Sep-82 10:43:43 PDT
RTTraceAndSweepImpl.mesa!1              9-Sep-82 10:42:04 PDT
RTTypesBasicExtensionImpl.bcd!1         7-Sep-82 14:19:13 PDT
RTTypesBasicExtensionImpl.mesa!1       27-May-82 16:29:22 PDT
RTTypesBasicImpl.bcd!1                  9-Sep-82 10:37:50 PDT
RTTypesBasicImpl.mesa!1                 7-Sep-82 16:58:55 PDT
RTZonesImpl.bcd!1                       9-Sep-82 10:38:01 PDT
RTZonesImpl.mesa!1                      7-Sep-82 16:11:59 PDT
RunsImpl.bcd!1                          9-Sep-82 10:38:13 PDT
```

```
  RunsImpl.mesa!1                        26-Mar-82 18:14:34 PST
  ZoneCleanupImpl.bcd!1                   7-Sep-82 14:21:00 PDT
  ZoneCleanupImpl.mesa!1                 27-May-82 16:41:15 PDT
  ZoneIndicesImpl.bcd!1                   9-Sep-82 10:38:16 PDT
  ZoneIndicesImpl.mesa!1                 20-May-82 13:05:35 PDT
Exports [Indigo]<PreCedar>RuntimeTypes>  ReleaseAs [Indigo]<Cedar>RuntimeTypes>

  RTFiles.bcd!1                           7-Sep-82 14:13:29 PDT
  RTFiles.mesa!1                         29-Jun-82 12:03:29 PDT
  RTModelPrivate.bcd!1                    4-Oct-82 16:46:05 PDT
  RTModelPrivate.mesa!1                   4-Oct-82 16:43:35 PDT
  RTSymbols.bcd!1                        23-Sep-82 10:43:07 PDT
  RTSymbols.mesa!1                       23-Sep-82  9:43:13 PDT
  RTSymbolsPrivate.bcd!1                  1-Sep-82 17:33:58 PDT
  RTSymbolsPrivate.mesa!1                 5-Aug-82 10:45:14 PDT
  RTTBridge.bcd!1                         9-Sep-82  7:50:08 PDT
  RTTBridge.mesa!1                        8-Sep-82 11:52:52 PDT
  RTTCache.bcd!1                          1-Sep-82 17:32:46 PDT
  RTTCache.mesa!1                        24-Jun-82 17:29:49 PDT
  RTTRemoteBridge.bcd!1                  10-Sep-82 11:24:04 PDT
  RTTRemoteBridge.mesa!1                 10-Sep-82 10:43:02 PDT
  RTTypes.bcd!1                           9-Sep-82  7:49:57 PDT
  RTTypes.mesa!1                          9-Sep-82  7:37:31 PDT
  RTTypesExtras.mesa!1                   15-Sep-82 17:46:37 PDT
  RTTypesExtras.bcd!1                    15-Sep-82 17:49:02 PDT

Directory [Indigo]<PreCedar>RuntimeTypes>  ReleaseAs [Indigo]<Cedar>RuntimeTypes>

  +RTT.bcd!3                              6-Oct-82 19:04:36 PDT
  RTT.config!1                            4-Oct-82 11:58:54 PDT
  RemoteRope.mesa!1                      10-Sep-82 14:30:37 PDT
  RemoteRope.bcd!1                       10-Sep-82 15:34:18 PDT
  RemoteRopeImpl.mesa!1                  10-Sep-82 14:42:05 PDT
  RemoteRopeImpl.bcd!1                   10-Sep-82 15:34:24 PDT
  RTGetSymbolsImpl.bcd!1                 29-Sep-82 16:44:25 PDT
  RTGetSymbolsImpl.mesa!1                29-Sep-82 16:43:23 PDT
  RTMiniModelImpl.Mesa!1                 24-Sep-82 10:41:18 PDT
  RTMiniModelImpl.bcd!1                   4-Oct-82 16:46:17 PDT
  RTModelSourceImpl.mesa!1                4-Oct-82 14:49:52 PDT
  RTModelSourceImpl.bcd!1                 4-Oct-82 16:47:43 PDT
  RTModelSectionImpl.mesa!1               4-Oct-82 16:50:33 PDT
  RTModelSectionImpl.bcd!1                4-Oct-82 16:51:04 PDT
  RTModelContextImpl.mesa!1               5-Oct-82 17:52:27 PDT
  RTModelContextImpl.bcd!1                5-Oct-82 17:53:57 PDT
  RTModelPrivateImpl.mesa!1               4-Oct-82 12:14:02 PDT
  RTModelPrivateImpl.bcd!1                4-Oct-82 16:47:30 PDT
  RTWalkSymbolsImpl.bcd!1                23-Sep-82 10:49:40 PDT
  RTWalkSymbolsImpl.mesa!1               27-Apr-82 11:52:22 PDT
  RTTDefaultImpl.Mesa!1                   1-Oct-82 13:55:28 PDT
  RTTDefaultImpl.bcd!1                    1-Oct-82 13:55:53 PDT
  RTTGFNameImpl.Mesa!1                   12-Aug-82 15:27:32 PDT
  RTTGFNameImpl.bcd!1                    17-Sep-82 12:15:43 PDT
  RTTSupportImpl.bcd!2                   11-Oct-82 14:06:44 PDT
  RTTSupportImpl.mesa!2                  11-Oct-82 14:06:04 PDT
  RTTypesPrivate.bcd!1                   17-Sep-82 12:14:42 PDT
  RTTypesPrivate.mesa!1                  17-Sep-82 11:49:35 PDT
  RTTypesRemotePrivate.bcd!1             10-Sep-82 15:36:53 PDT
  RTTypesRemotePrivate.mesa!1            10-Sep-82 15:24:50 PDT
  RTTCacheImpl.bcd!1                      7-Sep-82 14:17:29 PDT
  RTTCacheImpl.mesa!1                    28-Jun-82 13:34:47 PDT
  RTTypesImpl.bcd!2                      11-Oct-82 14:10:20 PDT
  RTTypesImpl.mesa!2                     11-Oct-82 14:09:45 PDT
  RTTypesBridgeImpl.bcd!1                 1-Oct-82 11:22:25 PDT
  RTTypesBridgeImpl.mesa!1                1-Oct-82 11:21:43 PDT
  RTTypedVariablesImpl.bcd!1              6-Oct-82 16:27:41 PDT
```

```
  RTTypedVariablesImpl.mesa!1              6-Oct-82 16:25:51 PDT
  RTTypedFramesImpl.bcd!1                  6-Oct-82 18:12:09 PDT
  RTTypedFramesImpl.mesa!1                 6-Oct-82 18:10:25 PDT
  RTTypesRemoteImpl.bcd!1                 17-Sep-82 12:17:53 PDT
  RTTypesRemoteImpl.mesa!1                15-Sep-82 17:22:26 PDT
  RTTypesRemotePrivateImpl.bcd!1           4-Oct-82 16:48:05 PDT
  RTTypesRemotePrivateImpl.mesa!1          4-Oct-82 16:42:21 PDT

Imports [Indigo]<Cedar>Top>ListsAndAtoms.df Of >
    Using [Atom.bcd, List.bcd]

Imports [Indigo]<PreCedar>TeleDebug>WorldVM.df Of >
    Using [WorldVM.bcd, WorldVM.Mesa, WVM.bcd]

Imports [Indigo]<Cedar>Top>IO.df Of >
    Using [IO.bcd, IO.Mesa]

Imports [Indigo]<Cedar>Top>TTYIO.df Of >
    Using [TTYIO.bcd, TTYIO.Mesa]

Imports [Indigo]<Cedar>Top>CedarSnapshot.df Of >
    Using [CedarSnapshot.bcd, CedarSnapshot.Mesa]

Imports [Indigo]<Cedar>Top>CIFS.df Of >
    Using [CIFS.bcd, CIFS.Mesa]

Imports [Indigo]<Cedar>Top>CompatibilityPackage.df Of >
    Using [Ascii.bcd, Directory.bcd, LongString.bcd, TTY.bcd]

Imports [Indigo]<Cedar>Pilot>MesaRuntime.df Of >
    Using [Processes.bcd]

Imports [Indigo]<Cedar>Top>PilotInterfaces.df Of >
    Using [TrapSupport.bcd, DeviceCleanup.bcd, Frame.bcd, PrincOpsRuntime.bcd,
       ProcessOperations.bcd, PSB.bcd, TimeStamp.bcd, SpecialSpace.bcd,
       CPSwapDefs.bcd, Environment.bcd, File.bcd, Heap.bcd, Inline.bcd,
       MiscAlpha.bcd, Mopcodes.bcd, PrincOps.bcd, Process.bcd, Runtime.bcd,
       SDDefs.bcd, Space.bcd, Transaction.bcd, Volume.bcd, CachedSpace.bcd,
       ProcessInternal.bcd, ProcessPriorities.bcd, RuntimeInternal.bcd,
       System.bcd]

Imports [Indigo]<Cedar>Top>Random.df Of >
    Using [RandomInt.bcd, RandomIntImpl.bcd]

Imports [Indigo]<Cedar>Top>Rigging.df Of >
    Using [ShowTime.bcd, Convert.bcd, ConvertUnsafe.bcd, Rope.bcd,
       RopeInline.bcd]

Imports [Indigo]<Cedar>Top>BCD.df Of >
    Using [BcdDefs.bcd, BcdOps.bcd, FileSegment.bcd, Literals.bcd,
       RCMap.bcd, RCMapOps.Bcd, RCMapBuilderImpl.Bcd, RTBcd.bcd,
       SymbolCache.bcd, SymbolPack.bcd, Symbols.bcd, SymbolSegment.bcd,
       SymbolTable.bcd, Table.bcd, Tree.bcd, TypeStrings.bcd, TypeStringsImpl.bcd,
       RTSD.bcd, Strings.bcd]

Imports [Indigo]<Cedar>Top>Loader.df Of >
    Using [PilotLoadStateOps.bcd, PilotLoadStateOps.Mesa, PilotLoadStateFormat.bcd,
       PilotLoadStateFormat.Mesa, Loader.bcd, Loader.Mesa, PilotLoadStatePrivate.bcd,
       PilotLoadStatePrivate.Mesa]

Imports [Indigo]<Cedar>Top>VersionMap.df Of >
    Using [VersionMap.bcd, VersionMap.Mesa]
```

# Appendix C - Modeller Interfaces

## *Modeller Interface*

Chapter 4 presented the interface to the modeller as seen by the user. That interface is implemented using the following interface. Some non-essential details have been omitted. "--" at the beginning of a line begins a comment line.

```
ModellerInterface: DEFINITIONS = {
   Object: TYPE = REF OpaqueObject;    -- opaque to the user

   -- returns an object that must be passed to other calls
   StartModelling: PROC[modelName: STRING] RETURNS[Object];

   -- updates model if fileName is mentioned and has changed
   Notice: PROC[obj: Object, fileName: STRING];

   -- compile any files mentioned in the model that need it
   -- if modules have been loaded, and UnLoadModelBcds
   -- has not been called, tries to compile for module replacement
   CompileModelSrcs: PROC[obj: Object];

   -- load object files for modules listed in model
   -- if modules are already loaded, tries to use module replacement
   LoadModelBcds: PROC[obj: Object];

   -- start execution of modules just loaded
   -- must be preceded by a LoadModelBcds
   StartModelBcds: PROC[obj: Object];

   -- unload the object files loaded by LoadModelBcds
   -- does nothing if no files are loaded
   UnLoadModelBcds: PROC[obj: Object];

   -- make a .modelBcd file for this model
   MakeModelBcds: PROC[obj: Object];

   -- equivalent to
   --       UnLoadModelBcds[obj]; CompileModelSrcs[obj];
   --       LoadModelBcds[obj]; StartModelBcds[obj];
   -- but pauses if there are errors
   -- because it calls UnLoadModelBcds, will not try module replacement
   Begin: PROC[obj: Object];

   -- equivalent to
   --       CompileModelSrcs[obj]; LoadModelBcds[obj];
   -- but pauses if there are errors
   -- because there is no call to UnLoad, will try module replacement
   Continue: PROC[obj: Object];

   -- store any changed files on remote file servers
   StoreBack: PROC[obj: Object];

   -- must be paired with StartModelling
   StopModelling: PROC[obj: Object];

   }.
```

## *RTModel Interface*

The RTModel interface is used by the debugger to obtain information about the program being debugged. Its design shows the power of combining inter- and intra-module relationships in one set of terminology and one set of data structures.

*Terminology*

A *section* is a machine-oriented representation of an object such as a procedure, program, or model. A section is composed of pointers to:

1. the source for this program (or procedure or model),
2. the parameters with which it was called, in cases where they are constant, (This includes INTERFACE parameters in the modelling language.)
3. the object code (may be NIL),
4. the types of the parameters to the program (or procedure or model),
5. the type of parameters returned,
6. the sections of the components that are contained within this section, and
7. the sections of the statically enclosing program unit.

An object file from the compiler or a .modelBcd file produced by the Modeller is an example of a section.

A *context* is an abstract object that changes when control is transferred from one section to another. A stack frame for the local variables of a procedure call (*local frame*) is a context. Calling one procedure from another changes contexts. The record that contains global data per module (*global frame*) is a context, established when the module is loaded and initialization is performed. The local variables in an SML procedure are also a context.

A *closure* is a list of contexts that describe a naming environment. The closure follows static links in the local frames and is used to handle variables at runtime.

A *tree* is a parse tree for a procedure, a module, or a model. The tree is connected to trees for contained objects, for example, the tree for a model is connected to the trees of the procedures declared in the module.

These are shown in Figure C.1.

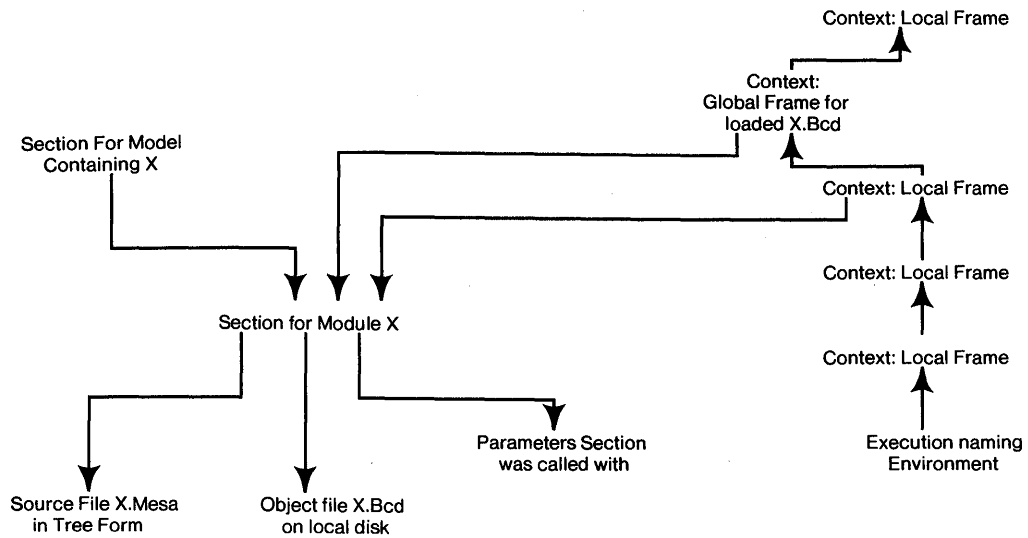



Figure C.1

```
-- RTModel.Mesa
RTModel: DEFINITIONS = {

    -- declarations of types
    Tree: TYPE = REF TreeObj;
    TreeObj: TYPE;

    Section: TYPE = REF SectionObj;
    SectionObj: TYPE;

    Context: TYPE = REF ContextObj;
    ContextObj: TYPE;

    -- gives the tree is which this tree appeared
    EnclosingTree: PROC[Tree] RETURNS[Tree];
    -- examine each of the subtrees this tree contains
    EnumSubTrees: PROC[Tree, PROC[Tree]];
    -- examine each of the copies of the tree that are loaded
    EnumSections: PROC[Tree, PROC[Section]];

    SectionToTree: PROC[Section] RETURNS[Tree];
    -- return the parameters to this section
    SectionParams: PROC[Section] RETURNS[REF ANY];
    -- return the section that loaded this section
```

```
EnclosingSection: PROC[Section] RETURNS[Section];
-- examine the sections that loading this section caused to be loaded
EnumSubSections: PROC[Section, PROC[Section]];
-- examine currently executing instances of this section
EnumContexts: PROC[Section, PROC[Context]];

ContextToSection: PROC[Context] RETURNS[Section];
-- return arguments that this was called with
ContextArgs: PROC[Context] RETURNS[REF ANY];
-- context that is statically outside this one
Encloser: PROC[Context] RETURNS[Context];
-- context that called this one
Invoker: PROC[Context] RETURNS[Context];
-- the contexts that this context has called
EnumInvokees: PROC[Context, PROC[Context]];
-- the contexts statically nested in this one that are still running
EnumEnclosees: PROC[Context, PROC[Context]];

}.
```

# Appendix D: Example Model

*An Example (Defaults)*

This model describes the BringOver program. First, we present the model with defaults, which reduces the complexity of the model to that of a simple CONFIGURATION module, and then expand the various defaulting rules to show the entire dependency information.

There are seven implementation modules within this model (CWFImpl, ComParseImpl, SubrImpl, STPSubrImpl, DFSubrImpl, DFParserImpl, BringOverImpl). All the rest are interface files.

Consider two models, PilotInterfaces.Model and BringOver.Model:

```
-- File: PilotInterfaces.Model!123, last edit September 7, 1982
[
  Ascii ~ @Ascii.Bcd;
  CIFS ~ @CIFS.Bcd;
  ConvertUnsafe ~ @ConvertUnsafe.Bcd;
  Date ~ @Date.Bcd;
  DCSFileTypes ~ @DCSFileTypes.Bcd;
  Directory ~ @Directory.Bcd;
  Environment ~ @Environment.Bcd;
  Exec ~ @Exec.Bcd;
  File ~ @File.Bcd;
  FileStream ~ @FileStream.Bcd;
  Heap ~ @Heap.Bcd;
  Inline ~ @Inline.Bcd;
  KernelFile ~ @KernelFile.Bcd;
  LongString ~ @LongString.Bcd;
  NameAndPasswordOps ~ @NameAndPasswordOps.Bcd;
  Process ~ @Process.Bcd;
  Rope ~ @Rope.Bcd;
  RopeInline ~ @RopeInline.Bcd;
  Runtime ~ @Runtime.Bcd;
  Segments ~ @Segments.Bcd;
  Space ~ @Space.Bcd;
  Storage ~ @Storage.Bcd;
  STP ~ @STP.Bcd;
  STPOps ~ @STPOps.Bcd;
  Stream ~ @Stream.Bcd;
  String ~ @String.Bcd;
  System ~ @System.Bcd;
  SystemInternal ~ @SystemInternal.Bcd;
  Time ~ @Time.Bcd;
  Transaction ~ @Transaction.Bcd;
  TTY ~ @TTY.Bcd;
  UserTerminal ~ @UserTerminal.Bcd;
];

-- File: BringOver.Model, last edit January 15, 1981
LET @PilotInterfaces.DF IN [
  BringOver: [CIFSImpl: CIFS,
         ConvertUnsafeImpl: ConvertUnsafe,
         DateImpl: Date,
         DirectoryImpl: Directory,
         ExecImpl: Exec,
         FileImpl: File,
         FileStreamImpl: FileStream,
         HeapImpl: Heap,
         InlineImpl: Inline,
         KernelFileImpl: KernelFile,
         LongStringImpl: LongString,
         NameAndPasswordOpsImpl: NameAndPasswordOps,
         ProcessImpl: Process,
         RopeImpl: Rope,
         RopeInlineImpl: RopeInline,
```

```
        RuntimeImpl: Runtime,
        SegmentsImpl: Segments,
        SpaceImpl: Space,
        StorageImpl: Storage,
        STPImpl: STP,
        STPOpsImpl: STPOps,
        StreamImpl: Stream,
        StringImpl: String,
        TimeImpl: Time,
        TransactionImpl: Transaction,
        TTYImpl: TTY,
        UserTerminalImpl: UserTerminal]
   -> [BringOverInterface: INTERFACE, BringOverCall: INTERFACE, BringOverCallImpl:
   BringOverCall,
   BringOverInterfaceImpl: BringOverInterface] ~ [
     CWF ~ @CWF.Bcd;
     CWFImpl ~ @CWFImpl.Bcd;
     ComParse ~ @ComParse;
     ComParseImpl ~ @ComParseImpl;
     Subr ~ @Subr;
     SubrImpl ~ @SubrImpl;
     STPSubr ~ @STPSubr;
     STPSubrImpl ~ @STPSubrImpl;
     DFSubr ~ @DFSubr;
     DFUser ~ @DFUser;
     DFSubrImplA ~ @DFSubrImpl;
     DFSubrImplB ~ @DFParserImpl;
     DFSubrImpl ~ (DFSubrImplA) PLUS (DFSubrImplB);
     BringOverInterface ~ @BringOverInterface;
     BringOverCall ~ @BringOverCall;
     [BringOverImpl: CONTROL, BringOverCallImpl: BringOverCall,
       BringOverInterfaceImpl: BringOverInterface] ~ @BringOverImpl
     ]
 ]
```

The top group of names are Pilot System definitions files, stored in a model called PilotInterfaces.Model. Each name, e.g., "Ascii", is given an interface as its value. The type of the name, e.g., ": INTERFACE Ascii", can be omitted since the modeller can look at the type of the value "@Ascii.Bcd" and determine it is an interface named "Ascii".

The second model is called BringOver.Model and begins "LET". The *LET* clause forces the contents of a list of names to be visible "without qualification". This way, the standard versions of the Pilot definitions files in PilotInterfaces.Model are represented compactly in a separate file. After the LET is a list of parameters to the model, which must be passed in (in this case, by the Pilot Loader). In this case, those parameters are simply instances of interfaces of the required INTERFACE. The body of the model lists, one per line, either definitions files local to the model (i.e., definitions files from the programmer) or implementations of these definitions files. The host and directory information and create date information has been omitted in this example, but must be present for the modeller to retrieve and store files. Also notice one implementor (BringOverImpl) exports an interface "CONTROL"; by convention, such an implementor is a module the modeller will START on command.

By comparison, the C/Mesa configuration module looks like:

```
-- BringOver.Config, last edit August 16, 1982 4:55 pm
-- the configuration for the BringOver program

BringOver: CONFIGURATION
IMPORTS CIFS, ConvertUnsafe, Date, Directory, Exec,
    File, FileStream, Heap, KernelFile, LongString,
    NameAndPasswordOps, Process, Rope, RopeInline, Runtime,
    Segments, Space, STP, STPOps, Storage, Stream,
    String, Time, Transaction, TTY, UserTerminal
EXPORTS BringOverInterface, BringOverCall
CONTROL BringOverImpl = {
    CWFImpl;
    ComParseImpl;
    SubrImpl;
    STPSubrImpl;
    DFSubrImpl;
    DFParserImpl;
    BringOverImpl;
}.
```

Why is the C/Mesa description shorter? There are two reasons:

1. In its body, the model mentions all user-owned modules, both interfaces and implementations. The configuration body mentions only the user's implementations.
2. The parameter list of the model ("[CIFSImpl: CIFS, ...") has some duplicate information that could be eliminated by a coercion in the SML language that mapped all interfaces in PilotInterfaces.Model into similarly named interface records. The parameter list would then be replaced by:

```
BringOver: [Inst: @PilotInterfaces.Model] ~ [
        LET Inst IN [
                ... body
                ]
        ]
```

which defines Inst to be a group of interface records with types that are the values defined in PilotInterfaces.Model.

*An Example (No Information Defaulted)*

In the previous version, we omitted all the argument lists following @-signs, since the modeller can supply for each parameter an actual with the same name. (We could not default the parameters if a module imported more than one instance of an interface.) We also omitted the type of the object when it could be inferred from the object on the right side of the "~". Here is a version that stores the entire description in one file and uses no defaults.

```
Ascii: INTERFACE ~ @Ascii.Bcd;
CIFS: INTERFACE ~ @CIFS.Bcd;
ConvertUnsafe: INTERFACE ~ @ConvertUnsafe.Bcd;
Date: INTERFACE ~ @Date.Bcd;
DCSFileTypes: INTERFACE ~ @DCSFileTypes.Bcd;
Directory: INTERFACE ~ @Directory.Bcd;
Environment: INTERFACE ~ @Environment.Bcd;
Exec: INTERFACE ~ @Exec.Bcd;
File: INTERFACE ~ @File.Bcd;
FileStream: INTERFACE ~ @FileStream.Bcd;
Heap: INTERFACE ~ @Heap.Bcd;
Inline: INTERFACE ~ @Inline.Bcd;
KernelFile: INTERFACE ~ @KernelFile.Bcd;
```

```
LongString: INTERFACE ~ @LongString.Bcd;
NameAndPasswordOps: INTERFACE ~ @NameAndPasswordOps.Bcd;
Process: INTERFACE ~ @Process.Bcd;
Rope: INTERFACE ~ @Rope.Bcd;
RopeInline: INTERFACE ~ @RopeInline.Bcd;
Runtime: INTERFACE ~ @Runtime.Bcd;
Segments: INTERFACE ~ @Segments.Bcd;
Space: INTERFACE ~ @Space.Bcd;
Storage: INTERFACE ~ @Storage.Bcd;
STP: INTERFACE ~ @STP.Bcd;
STPOps: INTERFACE ~ @STPOps.Bcd;
Stream: INTERFACE ~ @Stream.Bcd;
String: INTERFACE ~ @String.Bcd;
System: INTERFACE ~ @System.Bcd;
SystemInternal: INTERFACE ~ @SystemInternal.Bcd;
Time: INTERFACE ~ @Time.Bcd;
Transaction: INTERFACE ~ @Transaction.Bcd;
TTY: INTERFACE ~ @TTY.Bcd;
UserTerminal: INTERFACE ~ @UserTerminal.Bcd;
BringOver: [CIFSImpl: CIFS,
         ConvertUnsafeImpl: ConvertUnsafe,
         DateImpl: Date,
         DirectoryImpl: Directory,
         ExecImpl: Exec,
         FileImpl: File,
         FileStreamImpl: FileStream,
         HeapImpl: Heap,
         InlineImpl: Inline,
         KernelFileImpl: KernelFile,
         LongStringImpl: LongString,
         NameAndPasswordOpsImpl: NameAndPasswordOps,
         ProcessImpl: Process,
         RopeImpl: Rope,
         RopeInlineImpl: RopeInline,
         RuntimeImpl: Runtime,
         SegmentsImpl: Segments,
         SpaceImpl: Space,
         StorageImpl: Storage,
         STPImpl: STP,
         STPOpsImpl: STPOps,
         StreamImpl: Stream,
         StringImpl: String,
         TimeImpl: Time,
         TransactionImpl: Transaction,
         TTYImpl: TTY,
         UserTerminalImpl: UserTerminal]
  -> [BringOverInterface: INTERFACE, BringOverCall: INTERFACE, BringOverCallImpl:
  BringOverCall,
  BringOverInterfaceImpl: BringOverInterface] ~ [
    CWF: INTERFACE ~ @CWF.Bcd;
    CWFImpl: CWF ~ @CWFImpl.Bcd[HeapImpl, InlineImpl, LongStringImpl, TimeImpl];
    ComParse: INTERFACE ~ @ComParse.Mesa;
    ComParseImpl: ComParse ~ @ComParseImpl.Mesa[Ascii, ComParse,
                      Exec, Storage, String, TTY, ExecImpl, StorageImpl,
                      StringImpl, TTYImpl];
    Subr: INTERFACE ~ @Subr.Mesa[File, Space, Stream, TTY];
    SubrImpl: Subr ~ @SubrImpl.Mesa[Ascii, CWF, DCSFileTypes,
                      Directory, Environment, Exec, File, FileStream,
                      Heap, Inline, LongString, NameAndPasswordOps, Runtime,
                      Segments, Space, Stream, Subr, System, TTY, CWFImpl,
                      DirectoryImpl, ExecImpl, FileImpl, FileStreamImpl,
                      HeapImpl, InlineImpl, LongStringImpl,
                      NameAndPasswordOpsImpl, RuntimeImpl, SegmentsImpl,
                      SpaceImpl, StreamImpl, TTYImpl];
    STPSubr: INTERFACE ~ @STPSubr.Mesa[File, STP, Stream, System, TTY];
    STPSubrImpl: STPSubr ~ @STPSubrImpl.Mesa[
              CIFS, ConvertUnsafe, CWF, Date, DCSFileTypes, Directory,
              Environment, Exec, File, FileStream, Inline, LongString,
              NameAndPasswordOps, Process, Space, Storage, STP, STPOps,
              STPSubr, STPSubrExtras, Stream, String, Subr, TTY, UserTerminal,
              CIFSImpl, ConvertUnsafeImpl, CWFImpl, DateImpl, DirectoryImpl,
              ExecImpl, FileImpl, FileStreamImpl, InlineImpl, LongStringImpl,
```

```
        NameAndPasswordOpsImpl, ProcessImpl, SpaceImpl, STPImpl,
        STPOpsImpl, StorageImpl, StreamImpl, StringImpl, SubrImpl,
        UserTerminalImpl];
DFSubr: INTERFACE ~ @DFSubr.Mesa[File, Stream, TTY];
DFUser: INTERFACE ~ @DFUser.Mesa[DFSubr, TTY];
DFSubrImplA: DFSubr ~ @DFSubrImpl.Mesa[CWF, DFSubr,
                DFUser, Directory, Environment, Exec, Heap, Inline,
                LongString, Space, STPSubr, Stream, String, Subr,
                SystemInternal, TTY, CWFImpl, DFSubrImpl, DirectoryImpl,
                ExecImpl, HeapImpl, InlineImpl, LongStringImpl,
                SpaceImpl, STPSubrImpl, StreamImpl, StringImpl,
                SubrImpl, TTYImpl];
DFSubrImplB: DFSubr ~ @DFParserImpl.Mesa[CWF, Date, DFSubr,
                Exec, LongString, Stream, String, Subr, Time, CWFImpl,
                DateImpl, DFSubrImpl, ExecImpl, LongStringImpl,
                StreamImpl, StringImpl, SubrImpl, TimeImpl];
DFSubrImpl: DFSubr ~ (DFSubrImplA) PLUS (DFSubrImplB);
BringOverInterface: INTERFACE ~ @BringOverInterface.Mesa;
BringOverCall: INTERFACE ~ @BringOverCall.Mesa[Rope, TTY];
[BringOverImpl: CONTROL, BringOverCallImpl: BringOverCall,
BringOverInterfaceImpl: BringOverInterface] ~ @BringOverImpl.Mesa[
      BringOverCall, BringOverInterface, CIFS, ComParse, CWF,
      Date, DFSubr, Directory, Exec, File, FileStream, KernelFile,
      LongString, Rope, RopeInline, Runtime, Space, Storage,
      STP, STPSubr, STPSubrExtras, Stream, String, Subr, Time,
      TTY, CIFSImpl, ComParseImpl, CWFImpl, DateImpl, DFSubrImpl,
      DirectoryImpl, ExecImpl, FileStreamImpl, KernelFileImpl,
      LongStringImpl, RuntimeImpl, RopeImpl, RopeInlineImpl,
      SpaceImpl, StorageImpl, STPImpl, STPSubrImpl, STPSubrExtrasImpl,
      StreamImpl, StringImpl, SubrImpl, TimeImpl, TTYImpl]
]
```

# Appendix E: The Compiler Model

The Cedar Compiler is one of the largest programs in the Cedar system. There are 84 implementation modules and roughly 45,000 lines of Cedar code in the compiler. The two models below are being used to develop the Cedar compiler. The host and directory information and create times have been omitted.

The C/Mesa configuration for the Compiler at the end of this section is followed by a version of the compiler with no defaults. These models are included to show the size of models the System Modeller can handle. The only difference between these and the models in Appendix D is size.

*Compiler Model (with Defaults)*

```
-- BasicPilot.Model, September 7, 1982
[
  Ascii ~ @Ascii.bcd;
  DCSFileTypes ~ @DCSFileTypes.bcd;
  Directory ~ @Directory.bcd;
  Environment ~ @Environment.bcd;
  Exec ~ @Exec.bcd;
  ExecOps ~ @ExecOps.bcd;
  Feedback ~ @Feedback.bcd;
  File ~ @File.bcd;
  FileStream ~ @FileStream.bcd;
  FileTypes ~ @FileTypes.bcd;
  Format ~ @Format.bcd;
  Heap ~ @Heap.bcd;
  Inline ~ @Inline.bcd;
  KernelFile ~ @KernelFile.bcd;
  LongString ~ @LongString.bcd;
  MiscAlpha ~ @MiscAlpha.bcd;
  PrincOps ~ @PrincOps.bcd;
  Process ~ @Process.bcd;
  ProcessorFace ~ @ProcessorFace.bcd;
  Runtime ~ @Runtime.bcd;
  SDDefs ~ @SDDefs.bcd;
  Space ~ @Space.bcd;
  Stream ~ @Stream.bcd;
  String ~ @String.bcd;
  Strings ~ @Strings.bcd;
  System ~ @System.bcd;
  TemporarySpecialExecOps ~ @TemporarySpecialExecOps.bcd;
  Time ~ @Time.bcd;
  TimeStamp ~ @TimeStamp.bcd;
  Transaction ~ @Transaction.bcd;
  TTY ~ @TTY.bcd;
  UserTerminal ~ @UserTerminal.bcd;
  Volume ~ @Volume.bcd
  ]
-- Compiler.Model, 12-Aug-82 14:36:41 PDT
LET @BasicPilot.model IN [
  Compiler: [DirectoryImpl: Directory,
        ExecImpl: Exec,
        ExecOpsImpl1: ExecOps,
        FileImpl: File,
        FileStreamImpl: FileStream,
        HeapImpl: Heap,
        InlineImpl: Inline,
        KernelFileImpl: KernelFile,
        LongStringImpl: LongString,
        ProcessorFaceImpl: ProcessorFace,
        RuntimeImpl: Runtime,
        SpaceImpl: Space,
        StreamImpl: Stream,
        StringImpl: String,
        StringsImpl: Strings,
        TimeImpl: Time,
        TransactionImpl: Transaction,
```

```
      UserTerminalImpl: UserTerminal,
      VolumeImpl: Volume]
  -> [TemporarySpecialExecOpsImpl, ExecOpsImpl] ~ [
    -- interfaces and symbol tables
    RTSD ~ @RTSD[];
    Mopcodes ~ @Mopcodes.bcd;
    CommandUtil ~ @CommandUtil[];
    Table ~ @Table.mesa;
    Alloc ~ @Alloc.mesa;
    BcdDefs ~ @BcdDefs[];
    BcdOps ~ @BcdOps[];
    Symbols ~ @Symbols[];
    Literals ~ @Literals[];
    Tree ~ @Tree[];
    SymbolSegment ~ @SymbolSegment[];
    SymbolOps ~ @SymbolOps[];
    SymbolTable ~ @SymbolTable[];
    RCMap ~ @RCMap.mesa;
    RTBcd ~ @RTBcd[];
    FileSegment ~ @FileSegment[];
    FileParms ~ @FileParms[];
    LiteralOps ~ @LiteralOps[];
    TreeOps ~ @TreeOps[];
    SymLiteralOps ~ @SymLiteralOps[];
    Log ~ @Log[];
    Types ~ @Types[];
    Copier ~ @Copier[];
    CompilerOps ~ @CompilerOps[];
    CBinary ~ @CBinary.mesa;
    CompilerUtil ~ @CompilerUtil[];
    OSMiscOps ~ @OSMiscOps[];
    OSMiscOpsImpl: OSMiscOps ~ @OSMiscOpsImpl[];
    CharIO ~ @CharIO[];
    CharIOImpl: CharIO ~ @CharIOImpl[];
    Real ~ @IeeeFloat.mesa;
    RealImpl: Real ~ @IeeePack[];
    AllocImpl: Alloc ~ @AllocImpl[];
    [SymbolPack: INTERFACE, SymbolPackImpl: SymbolPack, SymbolOpsImpl1:
    SymbolOps] ~ @SymbolPack.mesa[];
    SymbolOpsImpl2: SymbolOps ~ @SymbolPackExt.mesa[];
    SymbolOpsImpl: SymbolOps ~ (SymbolOpsImpl1) PLUS (SymbolOpsImpl2);
    [SymbolPackB SymbolPack, SymbolPackImplB: SymbolPackB,
    SymbolOpsImplIgnore: SymbolOps] ~ @SymbolPack.mesa[];
    SymbolTableImpl: SymbolTable ~ @SymbolCache.mesa[SymbolPackB,
    SymbolPackImplB];
    TreeOpsImpl: TreeOps ~ @TreePack.mesa[];
    LiteralOpsImpl: LiteralOps ~ @LiteralPack[];
    TypesImpl: Types ~ @TypePack[];
    CopierImpl1: Copier ~ @SymbolCopier.mesa[];
    CopierImpl2: Copier ~ @FilePack.mesa[];
    CopierImpl: Copier ~ (CopierImpl1) PLUS (CopierImpl2);
    SymLiteralOpsImpl: SymLiteralOps ~ @SymLiteralPack.mesa[];
    TypeStrings ~ @TypeStrings[];
    TypeStringsImpl: TypeStrings ~ @TypeStringsImpl[];
    RCMapOps ~ @RCMapOps[];
    RCMapOpsImpl: RCMapOps ~ @RCMapBuilderImpl[];
    CompilerUtilImpl6: CompilerUtil ~ @ObjectOut[];
    --
    -- pass 1
    ParseTable ~ @ParseTable.mesa;
    P1 ~ @P1[];
    CBinaryImpl1: CBinary ~ @MesaTab.bcd;
    [CompilerUtilImpl1: CompilerUtil, P1Impl1: P1] ~ @Pass1[];
    P1Impl3: P1 ~ @Scanner[];
    P1Impl4: P1 ~ @Parser[];
    P1Impl2: P1 ~ @Pass1T[];
    P1Impl: P1 ~ (P1Impl1) PLUS (P1Impl2) PLUS (P1Impl3) PLUS (P1Impl4);
    --
    -- pass 2
    CompilerUtilImpl2: CompilerUtil ~ @Pass2[];
    --
    -- pass3
```

```
P3 ~ @P3[];
P3S ~ @P3S[];
[Pass3: INTERFACE, Pass3Impl: Pass3, CompilerUtilImpl3: CompilerUtil] ~
@Pass3[];
P3Impl1: P3 ~ @Pass3B[];
P3Impl2: P3 ~ @Pass3T[];
P3Impl3: P3 ~ @Pass3D[];
P3Impl4: P3 ~ @Pass3I[];
P3Impl5: P3 ~ @Pass3M[];
[P3Impl6: P3, P3SImpl1: P3S] ~ @Pass3S[];
P3Impl7: P3 ~ @Pass3V[];
[P3Impl8: P3, P3SImpl2: P3S] ~ @Pass3Xa[];
[P3Impl9: P3, P3SImpl3: P3S] ~ @Pass3Xb[];
P3Impl10: P3 ~ @Pass3Xc[];
P3Impl: P3 ~ (P3Impl1) PLUS (P3Impl2) PLUS (P3Impl3) PLUS (P3Impl4) PLUS
      (P3Impl5) PLUS (P3Impl6) PLUS (P3Impl7) PLUS (P3Impl8) PLUS
      (P3Impl9) PLUS (P3Impl10);
P3SImpl: P3S ~ (P3SImpl1) PLUS (P3SImpl2) PLUS (P3SImpl3);
CompilerUtilImpl7: CompilerUtil ~ @Pass3P[];
--
-- pass 4
P4 ~ @P4[];
[Pass4: INTERFACE, Pass4Impl: Pass4, CompilerUtilImpl4: CompilerUtil] ~
@Pass4[];
P4Impl1: P4 ~ @Pass4B[];
P4Impl2: P4 ~ @Pass4D[];
P4Impl3: P4 ~ @Pass4L[];
P4Impl4: P4 ~ @Pass4S[];
P4Impl5: P4 ~ @Pass4Ops[];
P4Impl6: P4 ~ @Pass4Xa[];
P4Impl7: P4 ~ @Pass4Xb[];
P4Impl8: P4 ~ @Pass4Xc[];
ReplOps ~ @ReplOps[];
ReplOpsImpl: ReplOps ~ @ReplPack[];
P4Impl: P4 ~ (P4Impl1) PLUS (P4Impl2) PLUS (P4Impl3) PLUS (P4Impl4) PLUS
      (P4Impl5) PLUS (P4Impl6) PLUS (P4Impl7) PLUS (P4Impl8);
--
-- pass 5 and 6
P5 ~ @P5[];
CodeDefs ~ @CodeDefs[];
P5F ~ @P5F[];
P5L ~ @P5L[];
P5S ~ @P5S[];
P5U ~ @P5U[];
PeepholeDefs ~ @PeepholeDefs[];
Stack ~ @Stack[];
Counting ~ @Counting[];
FOpCodes ~ @FOpCodes.mesa;
OpCodeParams ~ @OpCodeParams[];
OpTableDefs ~ @OpTableDefs[];
[Code: INTERFACE, CodeImpl: Code, CompilerUtilImpl5: CompilerUtil] ~ @Code[];
P5UImpl: P5U ~ @CgenUtil[];
[CodeDefsImpl2: CodeDefs, P5Impl1: P5] ~ @Temp[];
[P5LImpl1: P5L, CodeDefsImpl3: CodeDefs] ~ @VarUtils[];
[P5LImpl2: P5L, CodeDefsImpl4: CodeDefs] ~ @VarBasics[];
[P5LImpl3: P5L, CodeDefsImpl5: CodeDefs] ~ @VarMove[];
[P5Impl2: P5, P5SImpl1: P5S] ~ @Driver[];
OpTableDefsImpl: OpTableDefs ~ @OpTable[];
P5Impl3: P5 ~ @FOpTable[];
[CodeDefsImpl7: CodeDefs, P5SImpl2: P5S] ~ @Address[];
StackImpl: Stack ~ @StackImpl[];
[CodeDefsImpl9: CodeDefs, P5Impl4: P5, P5SImpl3: P5S] ~ @Flow[];
[CodeDefsImpl10: CodeDefs, P5Impl5: P5, P5SImpl4: P5S] ~ @Calls[];
[CodeDefsImpl11: CodeDefs, P5Impl6: P5, P5SImpl5: P5S] ~ @Store[];
[CountingImpl: Counting, CodeDefsImpl12: CodeDefs] ~ @CountingImpl[];
[CodeDefsImpl13: CodeDefs, P5Impl7: P5] ~ @Constructor[];
[CodeDefsImpl14: CodeDefs, P5Impl8: P5] ~ @Expression[];
[CodeDefsImpl15: CodeDefs, P5Impl9: P5] ~ @FlowExpression[];
[CodeDefsImpl16: CodeDefs, P5Impl10: P5] ~ @Statement[];
[CodeDefsImpl17: CodeDefs, P5Impl11: P5] ~ @Selection[];
[CodeDefsImpl18: CodeDefs, P5Impl12: P5] ~ @OutCode[];
[CodeDefsImpl19: CodeDefs, P5Impl13: P5, PeepholeDefsImpl1: PeepholeDefs] ~
```

```
@PeepholeQ[];
[P5Impl14: P5, PeepholeDefsImpl2: PeepholeDefs] ~ @PeepholeU[];
PeepholeDefsImpl3: PeepholeDefs ~ @PeepholeZ[];
[CodeDefsImpl20: CodeDefs, P5FImpl1: P5F] ~ @DJumps[];
[CodeDefsImpl21: CodeDefs, P5FImpl2: P5F] ~ @CrossJump[];
[CodeDefsImpl22: CodeDefs, P5Impl15: P5, P5FImpl3: P5F] ~ @Final[];
CodeDefsImpl: CodeDefs ~ (CodeDefsImpl2) PLUS (CodeDefsImpl3) PLUS
      (CodeDefsImpl4) PLUS (CodeDefsImpl5) PLUS (CodeDefsImpl7) PLUS
      (CodeDefsImpl9) PLUS (CodeDefsImpl10) PLUS (CodeDefsImpl11) PLUS
      (CodeDefsImpl12) PLUS (CodeDefsImpl13) PLUS (CodeDefsImpl14) PLUS
      (CodeDefsImpl15) PLUS (CodeDefsImpl16) PLUS (CodeDefsImpl17) PLUS
      (CodeDefsImpl18) PLUS (CodeDefsImpl19) PLUS (CodeDefsImpl20) PLUS
      (CodeDefsImpl21) PLUS (CodeDefsImpl22);
P5Impl: P5 ~ (P5Impl1) PLUS (P5Impl2) PLUS (P5Impl3) PLUS (P5Impl4) PLUS
      (P5Impl5) PLUS (P5Impl6) PLUS (P5Impl7) PLUS (P5Impl8) PLUS
      (P5Impl9) PLUS (P5Impl10) PLUS (P5Impl11) PLUS (P5Impl12) PLUS
      (P5Impl13) PLUS (P5Impl14) PLUS (P5Impl15);
P5FImpl: P5F ~ (P5FImpl1) PLUS (P5FImpl2) PLUS (P5FImpl3);
P5LImpl: P5L ~ (P5LImpl1) PLUS (P5LImpl2) PLUS (P5LImpl3);
P5SImpl: P5S ~ (P5SImpl1) PLUS (P5SImpl2) PLUS (P5SImpl3) PLUS
      (P5SImpl4) PLUS (P5SImpl5);
PeepholeDefsImpl: PeepholeDefs ~ (PeepholeDefsImpl1) PLUS
(PeepholeDefsImpl2)  PLUS  (PeepholeDefsImpl3);
PackageSymbols ~ @PackageSymbols[];
-- compiler control
[CompilerOpsImpl: CompilerOps, CompilerUtilImpl8: CompilerUtil] ~
@Sequencer[];
[ComData: INTERFACE, ComDataImpl: ComData] ~ @ComData[];
ErrorTable ~ @ErrorTable[];
LogImpl: Log ~ @LogPack[];
CBinaryImpl2: CBinary ~ @ErrorTab.bcd;
DebugTable ~ @DebugTable[];
CompilerUtilImpl9: CompilerUtil ~ @Debug[];
CBinaryImpl3: CBinary ~ @DebugTab.bcd;
CBinaryImpl: CBinary ~ (CBinaryImpl1) PLUS (CBinaryImpl2) PLUS
      (CBinaryImpl3);
CompilerUtilImpl: CompilerUtil ~ (CompilerUtilImpl1) PLUS
      (CompilerUtilImpl2) PLUS
      (CompilerUtilImpl3) PLUS (CompilerUtilImpl4) PLUS (CompilerUtilImpl5)
PLUS
      (CompilerUtilImpl6) PLUS (CompilerUtilImpl7) PLUS (CompilerUtilImpl8)
PLUS
      (CompilerUtilImpl9);
FileParmOps ~ @FileParmOps[];
FileParmOpsImpl: FileParmOps ~ @FileParmPack[];
[ExecOpsImpl2: ExecOps, TemporarySpecialExecOpsImpl: TemporarySpecialExecOps]
~ @Interface[];
ExecOpsImpl: ExecOps ~ (ExecOpsImpl2) THEN (ExecOpsImpl1);
CommandUtilImpl: CommandUtil ~ @CommandPack[];
TestC: CONTROL ~ @TestCompilerImpl[]
]
```

*Configuration for Compiler*

The C/Mesa configuration for the Compiler uses defaulting rules similar to the defaults in the modelling language.

```
-- Compiler.config, Compiler configuration
Compiler: CONFIG LINKS: CODE
    IMPORTS
      Directory, ExecOps, File, FileStream, Heap, Inline, KernelFile,
      LongString, ProcessorFace, Runtime, Space, String, Time,
      Transaction, Volume
    EXPORTS ExecOps, TemporarySpecialExecOps, CompilerOps
    CONTROL Interface = {

  SymCache: CONFIG
      IMPORTS File, Heap, LongString, Space, Strings, Transaction
      EXPORTS SymbolTable
```

```
    CONTROL SymbolCache = {
  SymbolPack;
  SymbolCache};

P1: CONFIG
    IMPORTS
      Alloc, ComData, CompilerUtil, CharIO, FileStream, LiteralOps,
      LongString, Real, Strings, SymbolOps, TreeOps
    EXPORTS CompilerUtil
    CONTROL Pass1 = {
  Pass1;
  Pass1T;
  Scanner;
  Parser};

P3: CONFIG
    IMPORTS
      Alloc, ComData, Copier, Log, LiteralOps, OSMiscOps,
      SymbolOps, SymbolPack, SymLiteralOps, TreeOps, Types
    EXPORTS CompilerUtil, Copier
    CONTROL Pass3 = {
  Pass3;
  SymbolCopier;
  Pass3B;
  Pass3T;
  Pass3D;
  Pass3I;
  Pass3M;
  Pass3S;
  Pass3V;
  Pass3Xa;
  Pass3Xb;
  Pass3Xc};

P4: CONFIG
    IMPORTS
      Alloc, ComData, CompilerUtil, Copier, Heap, Log, LongString, LiteralOps,
      Real, Strings, SymbolOps, SymLiteralOps, TreeOps, Types
    EXPORTS CompilerUtil
    CONTROL Pass4 = {
  Pass4;
  ReplPack;
  Pass4B;
  Pass4D;
  Pass4L;
  Pass4S;
  Pass4Ops;
  Pass4Xa;
  Pass4Xb;
  Pass4Xc};

P5: CONFIG
    IMPORTS
      Alloc, ComData, CompilerUtil, Counting, FileStream, Log, LiteralOps,
  OSMiscOps, Real, SymbolOps, SymLiteralOps, TreeOps
    EXPORTS CompilerUtil
    CONTROL Code = {
  Code;
  CgenUtil;
  Temp;
  VarUtils;
  VarBasics;
  VarMove;
  Driver;
  OpTable;
  FOpTable;
  Address;
  StackImpl;
  Flow;
  Calls;
  Store;
  CountingImpl;
```

```
        Constructor;
        Expression;
        FlowExpression;
        Statement;
        Selection;
        OutCode;
        PeepholeQ;
        PeepholeU;
        PeepholeZ;
        DJumps;
        CrossJump;
        Final};

    BcdOutput: CONFIG
        IMPORTS
          Alloc, ComData, FileStream, Heap, Inline, LiteralOps, LongString,

        OSMiscOps, SymbolOps, SymLiteralOps, TreeOps
      EXPORTS CompilerUtil = {
  ObjectOut;
  TypeStringsImpl;
  RCMapBuilderImpl};

-- Compiler specific system code
  OSMiscOpsImpl;
  CharIOImpl;
  IeeePack;
  AllocImpl;

-- Compiler utilities
  SymbolPack;
  SymbolPackExt;
  SymCache;
  TreePack;
  LiteralPack;
  SymLiteralPack;
  TypePack;
  FilePack;
  BcdOutput;
  MesaTab LINKS: FRAME;

-- Compiler passes
  P1;
  Pass2;
  P3;
  Pass3P;
  P4;
  P5;

-- Compiler control
  Sequencer;
  ComData;
  LogPack;
  ErrorTab LINKS: FRAME;
  Debug;
  DebugTab LINKS: FRAME;
  FileParmPack;
  Interface;
  CommandPack;

  }.
```

*Compiler Model (with No Defaults)*

```
-- BasicPilot.Model, September 7, 1982
[
  Ascii: INTERFACE ~ @Ascii.bcd;
  DCSFileTypes: INTERFACE ~ @DCSFileTypes.bcd;
  Directory: INTERFACE ~ @Directory.bcd;
  Environment: INTERFACE ~ @Environment.bcd;
```

```
  Exec: INTERFACE ~ @Exec.bcd;
  ExecOps: INTERFACE ~ @ExecOps.bcd;
  Feedback: INTERFACE ~ @Feedback.bcd;
  File: INTERFACE ~ @File.bcd;
  FileStream: INTERFACE ~ @FileStream.bcd;
  FileTypes: INTERFACE ~ @FileTypes.bcd;
  Format: INTERFACE ~ @Format.bcd;
  Heap: INTERFACE ~ @Heap.bcd;
  Inline: INTERFACE ~ @Inline.bcd;
  KernelFile: INTERFACE ~ @KernelFile.bcd;
  LongString: INTERFACE ~ @LongString.bcd;
  MiscAlpha: INTERFACE ~ @MiscAlpha.bcd;
  PrincOps: INTERFACE ~ @PrincOps.bcd;
  Process: INTERFACE ~ @Process.bcd;
  ProcessorFace: INTERFACE ~ @ProcessorFace.bcd;
  Runtime: INTERFACE ~ @Runtime.bcd;
  SDDefs: INTERFACE ~ @SDDefs.bcd;
  Space: INTERFACE ~ @Space.bcd;
  Stream: INTERFACE ~ @Stream.bcd;
  String: INTERFACE ~ @String.bcd;
  Strings: INTERFACE ~ @Strings.bcd;
  System: INTERFACE ~ @System.bcd;
  TemporarySpecialExecOps: INTERFACE ~ @TemporarySpecialExecOps.bcd;
  Time: INTERFACE ~ @Time.bcd;
  TimeStamp: INTERFACE ~ @TimeStamp.bcd;
  Transaction: INTERFACE ~ @Transaction.bcd;
  TTY: INTERFACE ~ @TTY.bcd;
  UserTerminal: INTERFACE ~ @UserTerminal.bcd;
  Volume: INTERFACE ~ @Volume.bcd
  ]

-- Compiler.Model, 12-Aug-82 14:36:41 PDT
LET @BasicPilot.model IN [
  Compiler: [DirectoryImpl: Directory,
        ExecImpl: Exec,
        ExecOpsImpl1: ExecOps,
        FileImpl: File,
        FileStreamImpl: FileStream,
        HeapImpl: Heap,
        InlineImpl: Inline,
        KernelFileImpl: KernelFile,
        LongStringImpl: LongString,
        ProcessorFaceImpl: ProcessorFace,
        RuntimeImpl: Runtime,
        SpaceImpl: Space,
        StreamImpl: Stream,
        StringImpl: String,
        StringsImpl: Strings,
        TimeImpl: Time,
        TransactionImpl: Transaction,
        UserTerminalImpl: UserTerminal,
        VolumeImpl: Volume]
    -> [TemporarySpecialExecOpsImpl, ExecOpsImpl] ~ [
      RTSD: INTERFACE ~ @RTSD.mesa[SDDefs];
      Mopcodes: INTERFACE ~ @Mopcodes.bcd;
      CommandUtil: INTERFACE ~ @CommandUtil.mesa[ExecOps, Stream, Strings];
      Table: INTERFACE ~ @Table.mesa;
      Alloc: INTERFACE ~ @Alloc.mesa;
      BcdDefs: INTERFACE ~ @BcdDefs.mesa[PrincOps, Table, TimeStamp];
      BcdOps: INTERFACE ~ @BcdOps.mesa[BcdDefs];
      Symbols: INTERFACE ~ @Symbols.mesa[PrincOps, Table, TimeStamp];
      Literals: INTERFACE ~ @Literals.mesa[Symbols, Table];
      Tree: INTERFACE ~ @Tree.mesa[Table, Literals, Symbols];
      SymbolSegment: INTERFACE ~ @SymbolSegment.mesa[Literals,
            Symbols, Table, TimeStamp, Tree];
      SymbolOps: INTERFACE ~ @SymbolOps.mesa[Alloc, Strings,
            Symbols, TimeStamp, Tree];
      SymbolTable: INTERFACE ~ @SymbolTable.mesa[FileSegment, SymbolPack];
      RCMap: INTERFACE ~ @RCMap.mesa;
      RTBcd: INTERFACE ~ @RTBcd.mesa[BcdDefs, RCMap, Symbols];
      FileSegment: INTERFACE ~ @FileSegment.mesa[File];
```

```
FileParms: INTERFACE ~ @FileParms.mesa[FileSegment, Strings, TimeStamp];
LiteralOps: INTERFACE ~ @LiteralOps.mesa[Alloc, Literals,  Strings, Symbols];
TreeOps: INTERFACE ~ @TreeOps.mesa[Alloc, Literals, Symbols, Tree];
SymLiteralOps: INTERFACE ~ @SymLiteralOps.mesa[Alloc, Literals, RTBcd,
Symbols,
       Tree];
Log: INTERFACE ~ @Log.mesa[Strings, Symbols, Tree];
Types: INTERFACE ~ @Types.mesa[SymbolTable, Symbols];
Copier: INTERFACE ~ @Copier.mesa[Alloc, FileParms, Strings,
      Symbols, SymbolTable, TimeStamp];
CompilerOps: INTERFACE ~ @CompilerOps.mesa[File, FileParms,
      Stream, Strings, TimeStamp];
CBinary: INTERFACE ~ @CBinary.mesa;
CompilerUtil: INTERFACE ~ @CompilerUtil.mesa[Alloc, CompilerOps,
      FileStream, Stream, Strings, Tree];
OSMiscOps: INTERFACE ~ @OSMiscOps.mesa[Environment, File,
      Strings, TimeStamp];
OSMiscOpsImpl: OSMiscOps ~ @OSMiscOpsImpl.mesa[DCSFileTypes,
            Directory, File, Inline, KernelFile, OSMiscOps,
            ProcessorFace, Runtime, Space, Time, TimeStamp,
            "-a-b-cj-ns", DirectoryImpl, InlineImpl, KernelFileImpl,
            ProcessorFaceImpl, RuntimeImpl, SpaceImpl, TimeImpl];
CharIO: INTERFACE ~ @CharIO.mesa[Format, Stream, Strings];
CharIOImpl: CharIO ~ @CharIOImpl.mesa[CharIO, Stream,
            Strings, "-a-b-cj-ns", StringsImpl, StreamImpl];
Real: INTERFACE ~ @IeeeFloat.mesa;
RealImpl: Real ~ @IeeePack.mesa[Inline, Real, "-a-b-cj-ns", InlineImpl];
AllocImpl: Alloc ~ @AllocImpl.mesa[Alloc, Environment,
            File, FileTypes, Heap, Inline, Runtime, Space, Volume,
            "-a-b-cj-ns", FileImpl, HeapImpl, InlineImpl, RuntimeImpl,
            SpaceImpl, VolumeImpl];
[SymbolPack: INTERFACE, SymbolPackImplA: SymbolPack, SymbolOpsImpl1:
SymbolOps] ~ @SymbolPack.mesa[
      Inline, Literals, Strings, Symbols, SymbolOps, SymbolSegment,
      TimeStamp, Tree, "-a-b-cj-ns", InlineImpl, StringsImpl];
SymbolOpsImpl2: SymbolOps ~ @SymbolPackExt.mesa[
            Alloc, Strings, Symbols, SymbolOps, SymbolPack,
            SymbolSegment, Tree, TreeOps, "-a-b-cj-ns", AllocImpl,
            StringsImpl, SymbolOpsImpl, TreeOpsImpl, SymbolPackImplA];
SymbolOpsImpl: SymbolOps ~ (SymbolOpsImpl1) PLUS (SymbolOpsImpl2);
[SymbolPackB: INTERFACE SymbolPack, SymbolPackImplB: SymbolPackB,
SymbolOpsImplIgnore: SymbolOps] ~ @SymbolPack.mesa[
      Inline, Literals, Strings, Symbols, SymbolOps, SymbolSegment,
      TimeStamp, Tree, "-a-b-cj-ns", InlineImpl, StringsImpl];
SymbolTableImpl: SymbolTable ~ @SymbolCache.mesa[
            Environment, File, FileSegment, Heap, Space, Symbols,
            SymbolPackB, SymbolSegment, SymbolTable, "-a-b-cj-ns",
            SymbolPackImplB, FileImpl, HeapImpl, SpaceImpl];
TreeOpsImpl: TreeOps ~ @TreePack.mesa[Alloc, Literals,
            Symbols, Tree, TreeOps, "-a-b-cj-ns", AllocImpl];
LiteralOpsImpl: LiteralOps ~ @LiteralPack.mesa[Alloc,
            Literals, LiteralOps, Strings, Symbols, "-a-b-cj-ns",
            AllocImpl, StringsImpl];
TypesImpl: Types ~ @TypePack.mesa[Strings, SymbolTable,
            Symbols, Types, "-a-b-cj-ns", StringsImpl];
CopierImpl1: Copier ~ @SymbolCopier.mesa[Alloc,
            Copier, Inline, LiteralOps, OSMiscOps, Strings,
            SymbolTable, Symbols, SymbolOps, SymbolPack, Tree,
            TreeOps, "-a-b-cj-ns", AllocImpl, CopierImpl, InlineImpl,
            LiteralOpsImpl, OSMiscOpsImpl, TreeOpsImpl, SymbolPackImplA,
            SymbolOpsImpl];
CopierImpl2: Copier ~ @FilePack.mesa[Alloc, Copier,
            FileParms, Strings, SymbolTable, Symbols, SymbolOps,
            SymbolPack, SymbolSegment, TimeStamp, "-a-b-cj-ns",
            AllocImpl, SymbolTableImpl, SymbolOpsImpl, SymbolPackImplA];
CopierImpl: Copier ~ (CopierImpl1) PLUS (CopierImpl2);
SymLiteralOpsImpl: SymLiteralOps ~ @SymLiteralPack.mesa[
            Alloc, ComData, Literals, LiteralOps, RTBcd, Strings,
            Symbols, SymbolOps, SymbolSegment, SymLiteralOps,
            Table, Tree, TreeOps, Types, "-a-b-cj-ns", AllocImpl,
            LiteralOpsImpl, SymbolOpsImpl, TreeOpsImpl, TypesImpl,
            ComDataImpl];
```

```
TypeStrings: INTERFACE ~ @TypeStrings.mesa[Symbols, SymbolTable];
TypeStringsImpl: TypeStrings ~ @TypeStringsImpl.mesa[
            Inline, Strings, Symbols, SymbolTable, TypeStrings,
            "-a-b-cj-ns", InlineImpl, StringsImpl];
RCMapOps: INTERFACE ~ @RCMapOps.mesa[RCMap, SymbolTable,
      Symbols];
RCMapOpsImpl: RCMapOps ~ @RCMapBuilderImpl.mesa[
            Inline, Table, Symbols, SymbolTable, Environment,
            RCMap, RCMapOps, "-a-b-cj-ns", InlineImpl];
CompilerUtilImpl6: CompilerUtil ~ @ObjectOut.mesa[
            Alloc, BcdDefs, ComData, CompilerUtil, Environment,
            FileStream, Heap, Inline, Literals, LiteralOps,
            OSMiscOps, PackageSymbols, RCMap, RCMapOps, RTBcd,
            Stream, Strings, Symbols, SymbolSegment, SymbolOps,
            SymLiteralOps, Table, Tree, TreeOps, TypeStrings,
            "-a-b-cj-ns", AllocImpl, FileStreamImpl, HeapImpl,
            InlineImpl, OSMiscOpsImpl, LiteralOpsImpl, RCMapOpsImpl,
            StreamImpl, StringsImpl, SymbolOpsImpl, SymLiteralOpsImpl,
            TreeOpsImpl, TypeStringsImpl, ComDataImpl];
ParseTable: INTERFACE ~ @ParseTable.mesa;
P1: INTERFACE ~ @P1.mesa[ParseTable, Stream, Strings, Symbols];
CBinaryImpl1: CBinary ~ @MesaTab.bcd;
[CompilerUtilImpl1: CompilerUtil, P1Impl1: P1] ~ @Pass1.mesa[
      Alloc, BcdDefs, ComData, CompilerUtil, LiteralOps, P1,
      Strings, Symbols, SymbolOps, Tree, "-a-b-cj-ns", AllocImpl,
      LiteralOpsImpl, P1Impl, SymbolOpsImpl, ComDataImpl];
P1Impl3: P1 ~ @Scanner.mesa[Ascii, CharIO, CompilerUtil,
            Environment, FileStream, LiteralOps, P1, ParseTable,
            Real, Stream, Strings, SymbolOps, "-a-b-cj-ns",
            CharIOImpl, CompilerUtilImpl, FileStreamImpl, LiteralOpsImpl,
            RealImpl, StreamImpl, StringsImpl, SymbolOpsImpl];
P1Impl4: P1 ~ @Parser.mesa[CharIO, CompilerUtil,
            P1, ParseTable, Stream, Strings, "-a-b-cj-ns", CharIOImpl,
            CompilerUtilImpl, P1Impl];
P1Impl2: P1 ~ @Pass1T.mesa[ComData, ParseTable,
            P1, Symbols, Tree, TreeOps, "-a-b-cj-ns", P1Impl,
            TreeOpsImpl, ComDataImpl];
P1Impl: P1 ~ (P1Impl1) PLUS (P1Impl2) PLUS (P1Impl3) PLUS (P1Impl4);
CompilerUtilImpl2: CompilerUtil ~ @Pass2.mesa[Alloc,
            ComData, CompilerUtil, Log, Symbols, SymbolOps,
            Tree, TreeOps, "-a-b-cj-ns", AllocImpl, LogImpl,
            SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
P3: INTERFACE ~ @P3.mesa[Alloc, Copier, Inline, Symbols, Tree];
P3S: INTERFACE ~ @P3S.mesa[P3, Symbols, Tree];
[Pass3: INTERFACE, Pass3Impl: Pass3, CompilerUtilImpl3: CompilerUtil] ~
@Pass3.mesa[
      Alloc, ComData, CompilerUtil, Copier, Log, P3, SymLiteralOps,
      Symbols, Tree, TreeOps, "-a-b-cj-ns", AllocImpl, CopierImpl,
      LogImpl, P3Impl, SymLiteralOpsImpl, TreeOpsImpl, ComDataImpl];
P3Impl1: P3 ~ @Pass3B.mesa[Alloc, ComData, Copier,
            LiteralOps, Log, OSMiscOps, P3, Strings, Symbols,
            SymbolOps, Tree, TreeOps, "-a-b-cj-ns", AllocImpl,
            CopierImpl, LiteralOpsImpl, LogImpl, OSMiscOpsImpl,
            P3Impl, SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
P3Impl2: P3 ~ @Pass3T.mesa[Alloc, ComData, P3, Symbols,
            SymbolOps, Tree, TreeOps, Types, "-a-b-cj-ns", P3Impl,
            SymbolOpsImpl, TreeOpsImpl, TypesImpl, ComDataImpl];
P3Impl3: P3 ~ @Pass3D.mesa[Alloc, ComData, Inline,
            Log, P3, Symbols, SymbolOps, Tree, TreeOps, "-a-b-cj-ns",
            InlineImpl, LogImpl, P3Impl, SymbolOpsImpl, TreeOpsImpl,
            ComDataImpl];
P3Impl4: P3 ~ @Pass3I.mesa[Alloc, ComData, Copier,
            Log, P3, P3S, Symbols, SymbolOps, SymLiteralOps,
            Tree, TreeOps, "-a-b-cj-ns", AllocImpl, CopierImpl,
            LogImpl, P3Impl, P3SImpl, SymLiteralOpsImpl, SymbolOpsImpl,
            TreeOpsImpl, ComDataImpl];
P3Impl5: P3 ~ @Pass3M.mesa[Alloc, ComData, Log,
            Pass3, P3, P3S, Strings, Symbols, SymbolOps, Tree,
            TreeOps, Types, "-a-b-cj-ns", AllocImpl, LogImpl,
            P3Impl, P3SImpl, SymbolOpsImpl, TreeOpsImpl, TypesImpl,
            ComDataImpl, Pass3Impl];
[P3Impl6: P3, P3SImpl1: P3S] ~ @Pass3S.mesa[
```

```
        Alloc, ComData, Log, Pass3, P3, P3S, SymLiteralOps, Symbols,
        SymbolOps, Tree, TreeOps, "-a-b-cj-ns", LogImpl, P3Impl,
        P3SImpl, SymLiteralOpsImpl, SymbolOpsImpl, TreeOpsImpl,
        ComDataImpl, Pass3Impl];
P3Impl7: P3 ~ @Pass3V.mesa[Alloc, ComData, Copier,
            Log, P3, P3S, Symbols, SymbolOps, Tree, TreeOps,
            "-a-b-cj-ns", CopierImpl, LogImpl, P3Impl, P3SImpl,
            SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
[P3Impl8: P3, P3SImpl2: P3S] ~ @Pass3Xa.mesa[
        Alloc, ComData, Copier, Log, P3, P3S, Symbols, SymbolOps,
        Tree, TreeOps, Types, "-a-b-cj-ns", CopierImpl, LogImpl,
        P3Impl, P3SImpl, SymbolOpsImpl, TreeOpsImpl, TypesImpl,
        ComDataImpl];
[P3Impl9: P3, P3SImpl3: P3S] ~ @Pass3Xb.mesa[
        Alloc, ComData, LiteralOps, Log, P3, P3S, SymLiteralOps,
        Symbols, SymbolOps, Tree, TreeOps, Types, "-a-b-cj-ns",
        LiteralOpsImpl, LogImpl, P3Impl, P3SImpl, SymLiteralOpsImpl,
        SymbolOpsImpl, TreeOpsImpl, TypesImpl, ComDataImpl];
P3Impl10: P3 ~ @Pass3Xc.mesa[Alloc, ComData, Copier,
            Log, P3, P3S, Symbols, SymbolOps, Tree, TreeOps,
            "-a-b-cj-ns", CopierImpl, LogImpl, P3Impl, P3SImpl,
            SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
P3Impl: P3 ~ (P3Impl1) PLUS (P3Impl2) PLUS (P3Impl3) PLUS (P3Impl4) PLUS
        (P3Impl5) PLUS (P3Impl6) PLUS (P3Impl7) PLUS (P3Impl8) PLUS
        (P3Impl9) PLUS (P3Impl10);
P3SImpl: P3S ~ (P3SImpl1) PLUS (P3SImpl2) PLUS (P3SImpl3);
CompilerUtilImpl7: CompilerUtil ~ @Pass3P.mesa[Alloc,
            ComData, CompilerUtil, Log, Symbols, SymbolOps,
            Tree, TreeOps, "-a-b-cj-ns", AllocImpl, LogImpl,
            SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
P4: INTERFACE ~ @P4.mesa[Alloc, BcdDefs, Inline, LiteralOps,
        Literals, Symbols, Tree];
[Pass4: INTERFACE, Pass4Impl: Pass4, CompilerUtilImpl4: CompilerUtil] ~
@Pass4.mesa[
        Alloc, ComData, CompilerUtil, P4, Symbols, Tree, TreeOps,
        "-a-b-cj-ns", AllocImpl, TreeOpsImpl, P4Impl, ComDataImpl];
P4Impl1: P4 ~ @Pass4B.mesa[Alloc, BcdDefs, BcdOps,
            ComData, CompilerUtil, Copier, Environment, Heap,
            Log, P4, Pass4, PrincOps, ReplOps, Strings, Symbols,
            SymbolOps, SymbolTable, SymLiteralOps, Tree, TreeOps,
            Types, "-a-b-cj-ns", AllocImpl, CompilerUtilImpl,
            CopierImpl, HeapImpl, LogImpl, P4Impl, ReplOpsImpl,
            StringsImpl, SymbolOpsImpl, SymLiteralOpsImpl, TreeOpsImpl,
            TypesImpl, ComDataImpl, Pass4Impl];
P4Impl2: P4 ~ @Pass4D.mesa[Alloc, ComData, PrincOps,
            Log, P4, Symbols, SymbolOps, Tree, TreeOps, "-a-b-cj-ns",
            LogImpl, P4Impl, SymbolOpsImpl, TreeOpsImpl, ComDataImpl];
P4Impl3: P4 ~ @Pass4L.mesa[Alloc, ComData, CompilerUtil,
            Log, P4, PrincOps, Symbols, SymbolOps, Tree, TreeOps,
            "-a-b-cj-ns", CompilerUtilImpl, LogImpl, SymbolOpsImpl,
            TreeOpsImpl, ComDataImpl];
P4Impl4: P4 ~ @Pass4S.mesa[Alloc, ComData, Log,
            LiteralOps, P4, Pass4, PrincOps, Symbols, SymbolOps,
            SymLiteralOps, Tree, TreeOps, "-a-b-cj-ns", LogImpl,
            LiteralOpsImpl, P4Impl, SymbolOpsImpl, SymLiteralOpsImpl,
            TreeOpsImpl, ComDataImpl, Pass4Impl];
P4Impl5: P4 ~ @Pass4Ops.mesa[Alloc, Literals, LiteralOps,
            Log, P4, Pass4, Real, Symbols, Tree, TreeOps, "-a-b-cj-ns",
            LiteralOpsImpl, LogImpl, P4Impl, RealImpl, TreeOpsImpl,
            Pass4Impl];
P4Impl6: P4 ~ @Pass4Xa.mesa[Alloc, ComData, Environment,
            Heap, Inline, Literals, LiteralOps, Log, P4, Pass4,
            Symbols, SymbolOps, Tree, TreeOps, Types, "-a-b-cj-ns",
            HeapImpl, InlineImpl, LogImpl, LiteralOpsImpl, P4Impl,
            SymbolOpsImpl, TreeOpsImpl, TypesImpl, ComDataImpl,
            Pass4Impl];
P4Impl7: P4 ~ @Pass4Xb.mesa[Alloc, ComData, Heap,
            LiteralOps, Log, P4, Pass4, Symbols, SymbolOps,
            SymLiteralOps, Tree, TreeOps, "-a-b-cj-ns", HeapImpl,
            LogImpl, LiteralOpsImpl, P4Impl, SymbolOpsImpl,
            SymLiteralOpsImpl, TreeOpsImpl, ComDataImpl, Pass4Impl];
P4Impl8: P4 ~ @Pass4Xc.mesa[Alloc, ComData, Environment,
```

```
            Heap, LiteralOps, Log, P4, Symbols, SymbolOps, SymLiteralOps,
            Tree, TreeOps, "-a-b-cj-ns", HeapImpl, LogImpl,
            LiteralOpsImpl, P4Impl, SymbolOpsImpl, SymLiteralOpsImpl,
            TreeOpsImpl, ComDataImpl];
ReplOps: INTERFACE ~ @ReplOps.mesa[Symbols, SymbolTable];
ReplOpsImpl: ReplOps ~ @ReplPack.mesa[ReplOps, Strings,
            SymbolTable, Symbols, Types, "-a-b-cj-ns", StringsImpl,
            TypesImpl];
P4Impl: P4 ~ (P4Impl1) PLUS (P4Impl2) PLUS (P4Impl3) PLUS (P4Impl4) PLUS
      (P4Impl5) PLUS (P4Impl6) PLUS (P4Impl7) PLUS (P4Impl8);
P5: INTERFACE ~ @P5.mesa[CodeDefs, Literals, Symbols, Tree];
CodeDefs: INTERFACE ~ @CodeDefs.mesa[Alloc, Environment,
      Literals, PrincOps, Symbols, SymbolSegment, Table];
P5F: INTERFACE ~ @P5F.mesa[CodeDefs];
P5L: INTERFACE ~ @P5L.mesa[CodeDefs, Environment, Symbols];
P5S: INTERFACE ~ @P5S.mesa[CodeDefs, Tree];
P5U: INTERFACE ~ @P5U.mesa[Alloc, CodeDefs, PackageSymbols, Symbols, Tree];
PeepholeDefs: INTERFACE ~ @PeepholeDefs.mesa[Alloc, CodeDefs,
      FOpCodes, PrincOps];
Stack: INTERFACE ~ @Stack.mesa[Alloc, CodeDefs, Symbols];
Counting: INTERFACE ~ @Counting.mesa[CodeDefs, Symbols, Tree];
FOpCodes: INTERFACE ~ @FOpCodes.mesa;
OpCodeParams: INTERFACE ~ @OpCodeParams.mesa[Environment,
      Mopcodes, PrincOps];
OpTableDefs: INTERFACE ~ @OpTableDefs.mesa[Environment];
[Code: INTERFACE, CodeImpl: Code, CompilerUtilImpl5: CompilerUtil] ~
@Code.mesa[
      CodeDefs, CompilerUtil, P5, Symbols, "-a-b-cj-ns", P5Impl];
P5UImpl: P5U ~ @CgenUtil.mesa[Alloc, Code, CodeDefs,
            ComData, FOpCodes, LiteralOps, OpTableDefs, P5,
            P5U, PackageSymbols, PrincOps, Runtime, Stack, SymbolOps,
            Symbols, Table, Tree, TreeOps, "-a-b-cj-ns", AllocImpl,
            ComDataImpl, CodeImpl, LiteralOpsImpl, OpTableDefsImpl,
            P5Impl, RuntimeImpl, StackImpl, SymbolOpsImpl, TreeOpsImpl];
[CodeDefsImpl2: CodeDefs, P5Impl1: P5] ~ @Temp.mesa[
      Alloc, Code, CodeDefs, ComData, FOpCodes, Log, P5, P5U,
      PrincOps, Stack, SymbolOps, Symbols, "-a-b-cj-ns", ComDataImpl,
      CodeImpl, P5UImpl, LogImpl, P5Impl, StackImpl, SymbolOpsImpl];
[P5LImpl1: P5L, CodeDefsImpl3: CodeDefs] ~ @VarUtils.mesa[
      Alloc, BcdDefs, Code, CodeDefs, Environment, Inline, LiteralOps,
      Literals, P5, P5L, P5U, PrincOps, Stack, SymbolOps, Symbols,
      "-a-b-cj-ns", CodeImpl, InlineImpl, LiteralOpsImpl, P5Impl,
      P5UImpl, P5LImpl, StackImpl, SymbolOpsImpl];
[P5LImpl2: P5L, CodeDefsImpl4: CodeDefs] ~ @VarBasics.mesa[
      Alloc, Code, CodeDefs, PrincOps, Environment, FOpCodes,
      Inline, LiteralOps, Literals, P5, P5L, P5U, Stack, Symbols,
      "-a-b-cj-ns", CodeImpl, LiteralOpsImpl, P5Impl, P5UImpl,
      P5LImpl, StackImpl];
[P5LImpl3: P5L, CodeDefsImpl5: CodeDefs] ~ @VarMove.mesa[
      Alloc, Code, CodeDefs, Environment, FOpCodes, Inline,
      Literals, OpCodeParams, P5L, P5U, PrincOps, Stack, Symbols,
      "-a-b-cj-ns", CodeImpl, InlineImpl, P5UImpl, P5LImpl,
      StackImpl];
[P5Impl2: P5, P5SImpl1: P5S] ~ @Driver.mesa[
      Alloc, Code, CodeDefs, ComData, FOpCodes, P5, P5L, P5S,
      P5U, PrincOps, Stack, SymbolOps, Symbols, Tree, TreeOps,
      "-a-b-cj-ns", AllocImpl, ComDataImpl, CodeImpl, CodeDefsImpl,
      P5Impl, P5LImpl, P5UImpl, StackImpl, SymbolOpsImpl, TreeOpsImpl];
OpTableDefsImpl: OpTableDefs ~ @OpTable.mesa[OpTableDefs,
            "-a-b-cj-ns"];
P5Impl3: P5 ~ @FOpTable.mesa[P5, "-a-b-cj-ns"];
[CodeDefsImpl7: CodeDefs, P5SImpl2: P5S] ~ @Address.mesa[
      Alloc, Code, CodeDefs, ComData, FOpCodes, Inline, P5,
      P5L, P5S, P5U, SymbolOps, Symbols, Tree, TreeOps, "-a-b-cj-ns",
      ComDataImpl, CodeImpl, InlineImpl, P5UImpl, P5LImpl, P5Impl,
      SymbolOpsImpl, TreeOpsImpl];
StackImpl: Stack ~ @StackImpl.mesa[Alloc, Code,
            CodeDefs, FOpCodes, P5, P5L, P5U, Stack, Symbols,
            "-a-b-cj-ns", CodeImpl, P5Impl, P5LImpl, P5UImpl];
[CodeDefsImpl9: CodeDefs, P5Impl4: P5, P5SImpl3: P5S] ~ @Flow.mesa[
      Alloc, Code, CodeDefs, Environment, FOpCodes, Literals,
      P5, P5L, P5S, P5U, PrincOps, SDDefs, Stack, Symbols, Tree,
```

```
        TreeOps, "-a-b-cj-ns", CodeImpl, P5UImpl, P5LImpl, P5Impl,
        TreeOpsImpl, StackImpl];
[CodeDefsImpl10: CodeDefs, P5Impl5: P5, P5SImpl4: P5S] ~ @Calls.mesa[
        Alloc, Code, CodeDefs, ComData, Counting, Environment,
        FOpCodes, Log, OpTableDefs, P5, P5L, P5S, P5U, PrincOps,
        RTSD, SDDefs, Stack, SymbolOps, Symbols, Tree, TreeOps,
        "-a-b-cj-ns", ComDataImpl, CodeImpl, CountingImpl, LogImpl,
        OpTableDefsImpl, P5Impl, P5LImpl, P5UImpl, StackImpl,
        SymbolOpsImpl, TreeOpsImpl];
[CodeDefsImpl11: CodeDefs, P5Impl6: P5, P5SImpl5: P5S] ~ @Store.mesa[
        Alloc, Code, CodeDefs, ComData, Counting, Environment,
        FOpCodes, P5, P5L, P5S, P5U, Stack, SymbolOps, Symbols,
        Tree, TreeOps, "-a-b-cj-ns", CodeImpl, ComDataImpl, CountingImpl,
        P5UImpl, P5LImpl, P5Impl, StackImpl, SymbolOpsImpl, TreeOpsImpl];
[CountingImpl: Counting, CodeDefsImpl12: CodeDefs] ~ @CountingImpl.mesa[
        Alloc, CodeDefs, ComData, Counting, FOpCodes, P5, P5L,
        P5U, RTSD, Stack, SymbolOps, Symbols, SymLiteralOps, Tree,
        TreeOps, "-a-b-cj-ns", ComDataImpl, P5Impl, P5LImpl, P5UImpl,
        StackImpl, SymbolOpsImpl, SymLiteralOpsImpl, TreeOpsImpl];
[CodeDefsImpl13: CodeDefs, P5Impl7: P5] ~ @Constructor.mesa[
        Alloc, Code, CodeDefs, ComData, Counting, Environment,
        FOpCodes, Inline, LiteralOps, Literals, P5, P5L, P5U,
        PrincOps, SDDefs, Stack, Symbols, SymbolOps, Tree, TreeOps,
        "-a-b-cj-ns", ComDataImpl, CodeImpl, CountingImpl, InlineImpl,
        LiteralOpsImpl, P5Impl, P5LImpl, P5UImpl, StackImpl, SymbolOpsImpl,
        TreeOpsImpl];
[CodeDefsImpl14: CodeDefs, P5Impl8: P5] ~ @Expression.mesa[
        Alloc, BcdDefs, Code, CodeDefs, ComData, Environment,
        FOpCodes, Inline, Literals, OpCodeParams, P5, P5L, P5S,
        P5U, PrincOps, Real, Stack, SymbolOps, Symbols, Tree,
        TreeOps, "-a-b-cj-ns", CodeImpl, ComDataImpl, InlineImpl,
        P5Impl, P5LImpl, P5SImpl, P5UImpl, RealImpl, StackImpl,
        SymbolOpsImpl, TreeOpsImpl];
[CodeDefsImpl15: CodeDefs, P5Impl9: P5] ~ @FlowExpression.mesa[
        Alloc, Code, CodeDefs, FOpCodes, P5, P5L, P5U, Stack,
        Symbols, Tree, TreeOps, "-a-b-cj-ns", CodeImpl, P5UImpl,
        P5LImpl, P5Impl, StackImpl, TreeOpsImpl];
[CodeDefsImpl16: CodeDefs, P5Impl10: P5] ~ @Statement.mesa[
        Alloc, Code, CodeDefs, ComData, FOpCodes, Log, P5, P5L,
        P5S, P5U, PrincOps, Stack, SymbolOps, Symbols, Tree, TreeOps,
        "-a-b-cj-ns", ComDataImpl, CodeImpl, P5UImpl, P5LImpl,
        P5Impl, P5SImpl, StackImpl, SymbolOpsImpl, TreeOpsImpl,
        LogImpl];
[CodeDefsImpl17: CodeDefs, P5Impl11: P5] ~ @Selection.mesa[
        Alloc, Code, CodeDefs, ComData, FOpCodes, P5, P5L, P5S,
        P5U, RTSD, Stack, SymbolOps, Symbols, SymLiteralOps, Tree,
        TreeOps, "-a-b-cj-ns", ComDataImpl, CodeImpl, P5UImpl,
        P5LImpl, P5Impl, P5SImpl, StackImpl, SymbolOpsImpl, SymLiteralOpsImpl,
        TreeOpsImpl];
[CodeDefsImpl18: CodeDefs, P5Impl12: P5] ~ @OutCode.mesa[
        Alloc, Code, CodeDefs, ComData, CompilerUtil, Environment,
        FileStream, FOpCodes, Inline, Literals, LiteralOps, Log,
        Mopcodes, OSMiscOps, P5, P5U, PrincOps, Stack, Stream,
        Symbols, SymbolOps, SymbolSegment, Table, "-a-b-cj-ns",
        ComDataImpl, CodeImpl, CompilerUtilImpl, FileStreamImpl,
        InlineImpl, LiteralOpsImpl, LogImpl, OSMiscOpsImpl, P5Impl,
        P5UImpl, StackImpl, StreamImpl, SymbolOpsImpl];
[CodeDefsImpl19: CodeDefs, P5Impl13: P5, PeepholeDefsImpl1: PeepholeDefs] ~
@PeepholeQ.mesa[
        Alloc, Code, P5U, CodeDefs, FOpCodes, Inline, OpCodeParams,
        P5, PeepholeDefs, SDDefs, "-a-b-cj-ns", CodeImpl, InlineImpl,
        P5UImpl, P5Impl, PeepholeDefsImpl];
[P5Impl14: P5, PeepholeDefsImpl2: PeepholeDefs] ~ @PeepholeU.mesa[
        Alloc, Code, CodeDefs, FOpCodes, Inline, Mopcodes, OpCodeParams,
        OpTableDefs, P5, P5U, PeepholeDefs, PrincOps, "-a-b-cj-ns",
        CodeImpl, InlineImpl, P5UImpl, OpTableDefsImpl, P5Impl];
PeepholeDefsImpl3: PeepholeDefs ~ @PeepholeZ.mesa[
                Alloc, Code, CodeDefs, ComData, FOpCodes, Log, MiscAlpha,
                Mopcodes, OpCodeParams, OpTableDefs, P5, PeepholeDefs,
                P5U, RTSD, SDDefs, "-a-b-cj-ns", CodeImpl, ComDataImpl,
                LogImpl, OpTableDefsImpl, P5Impl, P5UImpl, PeepholeDefsImpl];
[CodeDefsImpl20: CodeDefs, P5FImpl1: P5F] ~ @DJumps.mesa[
```

```
      Alloc, Code, CodeDefs, OpCodeParams, P5F, "-a-b-cj-ns",
      CodeImpl, P5FImpl];
[CodeDefsImpl21: CodeDefs, P5FImpl2: P5F] ~ @CrossJump.mesa[
      Alloc, Code, CodeDefs, OpTableDefs, P5F, P5U, PeepholeDefs,
      "-a-b-cj-ns", CodeImpl, OpTableDefsImpl, P5UImpl, P5FImpl,
      PeepholeDefsImpl];
[CodeDefsImpl22: CodeDefs, P5Impl15: P5, P5FImpl3: P5F] ~ @Final.mesa[
      Alloc, Code, CodeDefs, ComData, FOpCodes, Mopcodes, OpCodeParams,
      OpTableDefs, P5, P5F, P5U, PeepholeDefs, "-a-b-cj-ns",
      CodeImpl, ComDataImpl, OpTableDefsImpl, P5UImpl, P5Impl,
      P5FImpl, PeepholeDefsImpl];
CodeDefsImpl: CodeDefs ~ (CodeDefsImpl2) PLUS (CodeDefsImpl3) PLUS
      (CodeDefsImpl4) PLUS (CodeDefsImpl5) PLUS (CodeDefsImpl7) PLUS
      (CodeDefsImpl9) PLUS (CodeDefsImpl10) PLUS (CodeDefsImpl11) PLUS
      (CodeDefsImpl12) PLUS (CodeDefsImpl13) PLUS (CodeDefsImpl14) PLUS
      (CodeDefsImpl15) PLUS (CodeDefsImpl16) PLUS (CodeDefsImpl17) PLUS
      (CodeDefsImpl18) PLUS (CodeDefsImpl19) PLUS (CodeDefsImpl20) PLUS
      (CodeDefsImpl21) PLUS (CodeDefsImpl22);
P5Impl: P5 ~ (P5Impl1) PLUS (P5Impl2) PLUS (P5Impl3) PLUS (P5Impl4) PLUS
      (P5Impl5) PLUS (P5Impl6) PLUS (P5Impl7) PLUS (P5Impl8) PLUS
      (P5Impl9) PLUS (P5Impl10) PLUS (P5Impl11) PLUS (P5Impl12) PLUS
      (P5Impl13) PLUS (P5Impl14) PLUS (P5Impl15);
P5FImpl: P5F ~ (P5FImpl1) PLUS (P5FImpl2) PLUS (P5FImpl3);
P5LImpl: P5L ~ (P5LImpl1) PLUS (P5LImpl2) PLUS (P5LImpl3);
P5SImpl: P5S ~ (P5SImpl1) PLUS (P5SImpl2) PLUS (P5SImpl3) PLUS
      (P5SImpl4) PLUS (P5SImpl5);
PeepholeDefsImpl: PeepholeDefs ~ (PeepholeDefsImpl1) PLUS
    (PeepholeDefsImpl2)  PLUS  (PeepholeDefsImpl3);
PackageSymbols: INTERFACE ~ @PackageSymbols.mesa[PrincOps,
      Symbols, SymbolSegment, Table];
[CompilerOpsImpl: CompilerOps, CompilerUtilImpl8: CompilerUtil] ~
  @Sequencer.mesa[
      CBinary, Alloc, CharIO, CompilerOps, CompilerUtil, ComData,
      Copier, File, FileParmOps, FileStream, LiteralOps, Log,
      OSMiscOps, Stream, Strings, SymLiteralOps, SymbolPack,
      SymbolOps, SymbolSegment, SymbolTable, Time, TimeStamp,
      Tree, TreeOps, "-a-b-cj-ns", AllocImpl, CBinaryImpl, CharIOImpl,
      CompilerUtilImpl, CopierImpl, FileStreamImpl, FileParmOpsImpl,
      LogImpl, LiteralOpsImpl, OSMiscOpsImpl, StreamImpl, SymLiteralOpsImpl,
      SymbolOpsImpl, SymbolTableImpl, StringsImpl, TimeImpl,
      TreeOpsImpl, SymbolPackImplA, ComDataImpl];
[ComData: INTERFACE, ComDataImpl: ComData] ~ @ComData.mesa[
      Alloc, BcdDefs, BcdOps, FileParms, OSMiscOps, Symbols,
      SymbolSegment, SymbolTable, Strings, Tree, "-a-b-cj-ns"];
ErrorTable: INTERFACE ~ @ErrorTable.mesa[Log, Tree];
LogImpl: Log ~ @LogPack.mesa[Alloc, CharIO, ComData,
             CompilerUtil, ErrorTable, FileStream, LiteralOps,
             Log, Stream, Strings, Symbols, SymbolOps, Tree,
             TreeOps, "-a-b-cj-ns", AllocImpl, CharIOImpl, CompilerUtilImpl,
             FileStreamImpl, LiteralOpsImpl, SymbolOpsImpl, TreeOpsImpl,
             ComDataImpl];
CBinaryImpl2: CBinary ~ @ErrorTab.bcd;
DebugTable: INTERFACE ~ @DebugTable.mesa[Symbols, Tree];
CompilerUtilImpl9: CompilerUtil ~ @Debug.mesa[Alloc,
             BcdDefs, CharIO, CompilerUtil, DebugTable, Literals,
             LiteralOps, Strings, Stream, Symbols, SymbolOps,
             Tree, TreeOps, "-a-b-cj-ns", AllocImpl, CharIOImpl,
             CompilerUtilImpl, LiteralOpsImpl, SymbolOpsImpl,
             TreeOpsImpl];
CBinaryImpl3: CBinary ~ @DebugTab.bcd;
CBinaryImpl: CBinary ~ (CBinaryImpl1) PLUS (CBinaryImpl2) PLUS
      (CBinaryImpl3);
CompilerUtilImpl: CompilerUtil ~ (CompilerUtilImpl1) PLUS
      (CompilerUtilImpl2) PLUS
      (CompilerUtilImpl3) PLUS (CompilerUtilImpl4) PLUS (CompilerUtilImpl5)
PLUS
      (CompilerUtilImpl6) PLUS (CompilerUtilImpl7) PLUS (CompilerUtilImpl8)
PLUS
      (CompilerUtilImpl9);
FileParmOps: INTERFACE ~ @FileParmOps.mesa[CommandUtil,
      File, FileParms, Strings];
FileParmOpsImpl: FileParmOps ~ @FileParmPack.mesa[
```

```
            BcdDefs, BcdOps, CommandUtil, File, FileParms, FileParmOps,
            FileSegment, OSMiscOps, Space, Strings, SymbolTable,
            TimeStamp, "-a-b-cj-ns", CommandUtilImpl, OSMiscOpsImpl,
            SpaceImpl, StringsImpl, SymbolTableImpl];
[ExecOpsImpl2: ExecOps, TemporarySpecialExecOpsImpl: TemporarySpecialExecOps]
~ @Interface.mesa[
      CharIO, CommandUtil, CompilerOps, ExecOps, Feedback, File,
      FileParms, FileParmOps, FileStream, Heap, Inline, OSMiscOps,
      Stream, String, Strings, TemporarySpecialExecOps, Time,
      TimeStamp, "-a-b-cj-ns", CharIOImpl, CommandUtilImpl,
      CompilerOpsImpl, ExecOpsImpl, FileStreamImpl, FileParmOpsImpl,
      HeapImpl, InlineImpl, OSMiscOpsImpl, StreamImpl, StringImpl,
      StringsImpl, TimeImpl];
ExecOpsImpl: ExecOps ~ (ExecOpsImpl2) THEN (ExecOpsImpl1);
CommandUtilImpl: CommandUtil ~ @CommandPack.mesa[
            Ascii, CharIO, CommandUtil, Heap, Stream, Strings,
            "-a-b-cj-ns", CharIOImpl, HeapImpl, StringsImpl];
TestC: CONTROL ~ @TestCompilerImpl.mesa[Ascii, Exec,
            ExecOps, Feedback, Heap, TemporarySpecialExecOps,
            UserTerminal, "-a-b-cj-ns", ExecImpl, HeapImpl,
            TemporarySpecialExecOpsImpl, UserTerminalImpl]
```