An Approach to Transforming Systems for Execution on Cloud Computing Systems

● Yoshiharu Maeda ● Manabu Kamimura ● Keisuke Yano

As business digitalization continues to accelerate, adapting existing enterprise systems to changing business practices and advances in information and communications technology (ICT) has become a significant problem. Not only must current systems be virtualized for execution on cloud computing systems, but software must also be restructured with higher flexibility to meet expanding business requirements, such as coordination with other services. However, it is typically not feasible to re-implement an entire system due to the high cost and risk of system malfunction. In this paper, we propose an approach to transforming a system in order to enhance its flexibility and expandability. Our approach works by extracting each part of the system individually, analyzing its characteristics, and identifying an appropriate implementation strategy based on those characteristics. Three techniques are used to support this approach. First, the structure of the system and the relationships between functions are visualized by analyzing the program files. Next, the business logic complexity, update frequency, etc. of each program is characterized. The feature values obtained are assigned to the heights of their corresponding structures on a software map and are used to characterize and prioritize functions or programs. Finally, the identified functions are analyzed using symbolic execution, and the rules and calculation methods used in the business are extracted as decision tables in a readable format. This approach enables an existing system, the system of record (SoR), to be transformed by extracting its features and identifying the best solution for each feature, such as defining it as a service, using it with a business rules management system (BRMS), or using the program as is.

1. Introduction

As information and communications technology (ICT), such as cloud and mobile computing, continues to evolve, adapting existing enterprise systems to changing business practices and new forms of ICT has become a significant problem. Infrastructure as a service (IaaS), an option for migrating enterprise systems that uses virtualization, can reduce hardware and host computer operation costs. However, improvements in system flexibility are limited because the programs and architecture of the system are left as is. Both the hardware and software must be continuously enhanced to provide the flexibility needed to support various types of changes over a long period of time.

Modernizing systems presents challenges different from those faced when developing a new system. These challenges include (1) program complication due to the accumulation of partial and irregular updates, (2) unavailable documentation describing the overall structure, including all updates and changes, although some records may be available, and (3) few developers with extensive knowledge of the current system.

As a result, many programs have become "black boxes" due to limited or no knowledge of their core functions. Since clients typically request that functions provided by their current system also be provided in the modernized system, the functions must be specified and their implementation clarified. Therefore, developers must take into account current functions as part of the modernization process, which differs from developing a system from scratch. The implementation method is determined after the specifications of the current functions are clarified.[1)]

This paper presents an approach developed by

Fujitsu Laboratories for extracting partial components from a complicated system and determining the best solution for transforming each one on the basis of its characteristics. It also presents a technique for analyzing program files to facilitate their use in the modernized system, thereby overcoming the difficulties mentioned above.

2. An approach for identifying an appropriate implementation

Since modernizing the entire enterprise system may be very costly and carry a high risk of failure, we propose dividing the system into functional parts and transforming each part into the best possible implementation on the basis of its characteristics (**Figure 1**). For example, parts performing common business operations can be transformed into a program product or software as a service (SaaS), and parts that may not need to be changed for a long time can continue to be used as they are. Parts performing complicated business operations and parts requiring frequent changes can be transformed into a business rule management system (BRMS), which clarifies business rules and supports high flexibility for business changes. Parts with high reusability can be transferred into services by using an application programming interface (API), which enables other systems or services to connect to them.

In the following section, we will introduce our approach and describe the three transformation stages: generating a software map, quantifying the complexity of the business logic, and creating decision tables from the programs.

3. Visualize entire system structure using a software map

The first hurdle in transforming software assets that have become a black box is determining how to divide them into parts performing different functions and having individual characteristics. This is difficult because the implemented functions are interconnected by program dependencies, such as subroutine calls and type references. To solve this problem, we have developed a technique for automatically creating a software map that provides a bird's-eye view of the system and extracts the parts that perform the major functions of

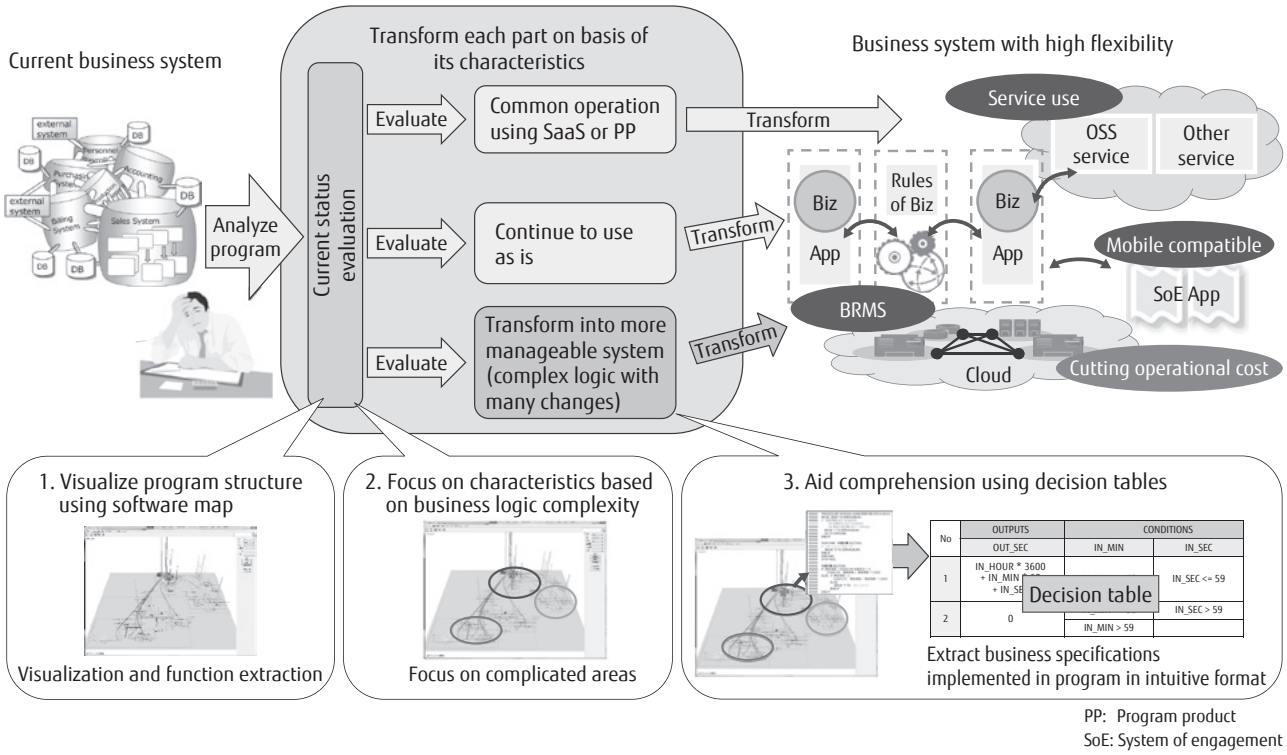

Figure 1
Approach to transforming current system for enhanced flexibility.

the system.[2),3)]

The software map technique takes the program files as input and analyzes them as a graph structure in order to extract clusters of tightly coupled programs (in terms of their dependencies) as functions of the system. The extracted clusters are laid out on a virtual surface to create a map-like visualization (**Figure 2**). Automatic extraction of software functions has hitherto been difficult because omnipresent modules, such as logging functions, connect various functions and obscure their distinctions.

The software map technique enables a software system to be divided into its functions automatically by assigning smaller weights to dependencies related to omnipresent modules. In the software map visualization, programs are represented as structures in a city block. Each city block corresponds to a software cluster and its corresponding function, meaning the number of city blocks represents the number of distinct implemented functions in the system. In the map-like visualization, the distance between city blocks reflects the distance between the corresponding functions. These characteristics provide a bird's-eye view of the entire system and the relationships between its functions (Figure 2).

4. Prioritize functions for transformation by using business logic complexity

The next challenge is visualizing the characteristics of each extracted functional part to enable determination of how to transform each part. The software map technique supports visualization of the characteristics of each program by assigning heights, colors, and shapes to the buildings that represent the program. This visualization technique provides intuitive information to the analyst and enables the features of the current system to be captured.

The characteristics of the program are the software metrics (e.g., complexity of the program, lines of code), the update records or frequencies, the quality information (e.g., bug reports), the operation logs, and the directory structures in the program. The characteristics to be visualized are selected and can be freely swapped for different characteristics. For example, when deciding which functions should remain as they are in accordance with the rate of change in the business, the program update frequencies are assigned as the building heights, and buildings with low heights are focused on as candidates for ones to remain as they are. The complexity of the process logic implemented in a program is referred to here as "program complexity." It can

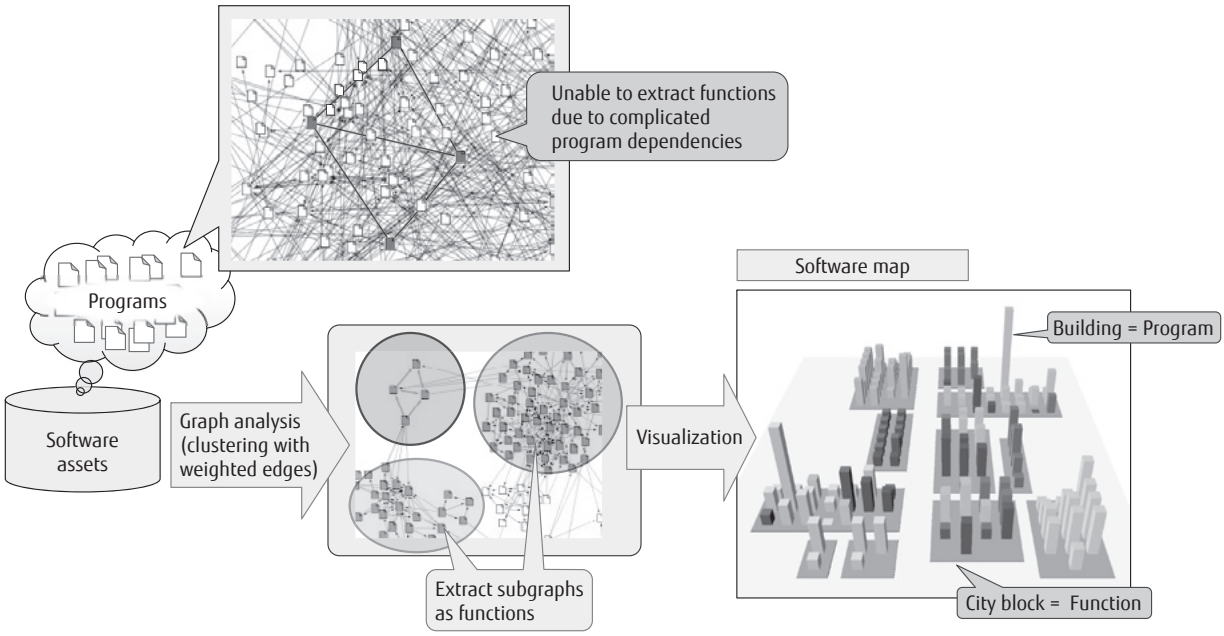

Figure 2
Generation of software map and example.

be calculated by counting the number of IF statements in the program or by counting the number of paths in the program structure. It can also be calculated by counting all the statements in the program.

There are two types of logic in a program: business logic (e.g., logic for calculating fees based on contracts) and control logic (e.g., logic for accessing databases). It is problematic if the complexity of a program does not match the complexity of a business. We propose a metric for the complexity of business logic, which quantifies the complexity of the original business logic by identifying the business logic in the program.[4] By using this "business logic complexity metric," an analyst can identify functions containing complicated business logic as well as those with a large number of updates, and select the ones to implement using a BRMS (**Figure 3**).

Business logic complexity is based on the assumption that statements that directly manipulate input or output data are related to business logic. Check logic (e.g., checking the input character type) and database access statements are excluded because these statements are less likely to be related to business logic. The feature value of complexity is calculated on the basis of the size of the decision table with statements related to business logic. Generating a precise decision table is expensive, as will be described in the following section. Thus, business logic complexity is calculated using an approximate size decision table derived from "the number of variables in the condition statements," "the number of cases," and "the number of variables in the computation statements." This approximation enables the value of business logic complexity of a program file to be quickly calculated even for a program with tens of thousands of lines of code. The characteristics of large systems can thus be captured.

Figure 3 presents an example of visualizing the characteristics of a function block by assigning a feature value (e.g., business logic complexity) to the height of a building on the software map. Comparing these feature values enables programmers to focus on the most significant functions and prioritize functions for transformation.

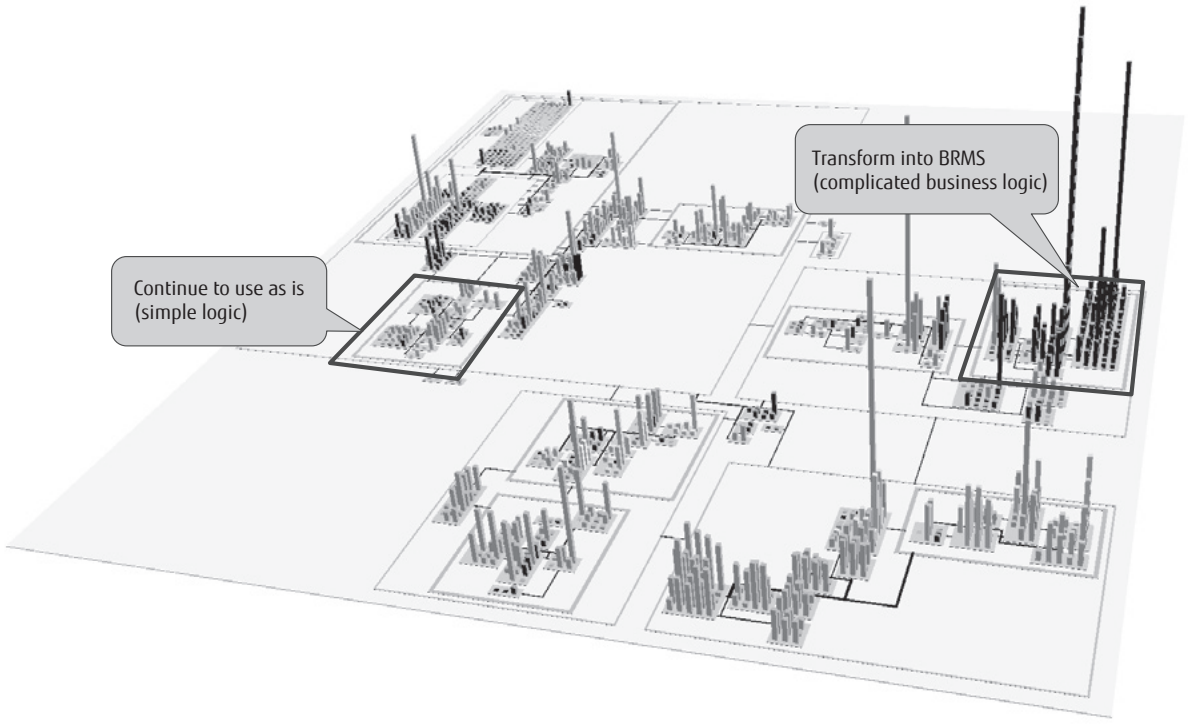

Figure 3
Example of transforming functions on basis of business complexity.

5. Create decision table for program comprehension

Using the aforementioned techniques, we specify the functional parts of the enterprise system as transitional targets and the program groups that comprise the parts. The next problem is to analyze the business logic of the specified programs in detail. To facilitate this, we developed a technique for extracting the business logic coded into a program, such as the rules and calculations for the business, in the form of a decision table. A decision table shows what values the program will output on the basis of the inputs (**Figure 4**).

The output values of the program are displayed in the output column of the decision table using only the input values and constants, with all internal variables removed. The conditions related to the outputs are shown in the conditions columns in a logically simplified form, rather than as extensive lists of conditions expressed by IF statements in the programs. Control structures, such as the subroutine calls and GO TO statements in the program, are simulated and removed from the decision table. These features enable the analyst to easily comprehend the business logic coded into the program by using a simple decision table.

A decision table is created by extracting and summarizing all the execution paths through the program. This analysis is performed using symbolic execution,[5] an analysis technique that extracts executable paths by setting variable symbolic values (not specific values such as 10 or ABC) as the inputs for the program and simulating the execution of the program. In a specially developed computational environment that can handle both symbolic and specific values, the symbolic and specific values are referenced and updated in accordance with statements in the program, and control statements, such as LOOP statements, are executed exactly as described in the program. Particularly when the true–false value of the condition for an IF statement cannot be evaluated because the condition includes symbolic values, the executable branch of the IF statement is selected in accordance with the satisfiability of the condition. Both THEN and ELSE branches in IF statements are evaluated as executables if the original condition and the negated condition are both satisfiable. For example, assume that the THEN branch is selected as a candidate path. Then, when the path reaches a stop statement or the final statement in the program, the path backtracks to the IF statement, and

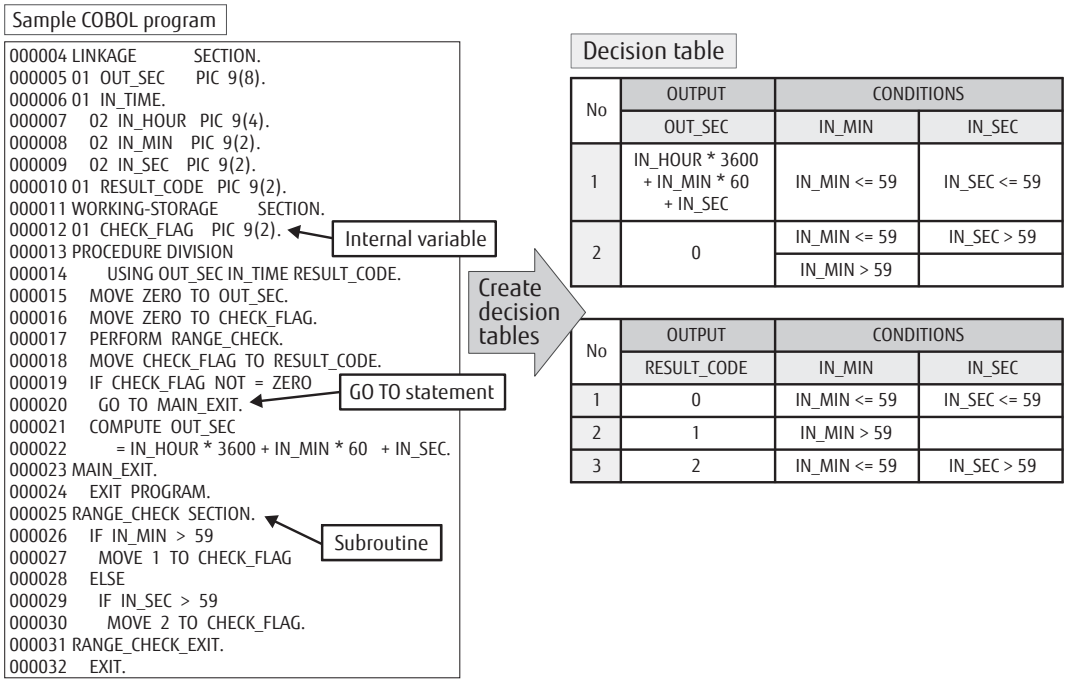

Figure 4
Creation of decision tables containing program outputs.

the ELSE branch is selected as the candidate path. In this manner, the executable paths are extracted using symbolic execution. Here, a condition is satisfiable if it can be made true by assigning appropriate specific values to the symbolic values in the condition. Generally, the satisfiability of a condition can be judged by using a tool called the Satisfiable Modulo Theories (SMT) solver.

Although symbolic execution has been studied academically since the 1970s, the development and application of practical tools was very slow until the performance of computers and the SMT solver accelerated in the 21st century. We developed a symbolic execution tool called SEA4COBOL (Symbolic Executing Analyzer for COBOL)[5] and applied it to test case generation for COBOL programs.[6] Another application of SEA4COBOL is the creation of decision tables based on COBOL programs. Although all the paths through a program must be extracted, it is difficult to apply symbolic execution to programs in real-world enterprise systems because the number of paths through a program increases rapidly with the size of the program and the number of conditional branches.

We have developed a three-step technique for overcoming this problem. First, the program is divided into processing blocks, each with a feasible number of paths, by analyzing the structure of the program. Next, each block is analyzed using symbolic execution, and a decision table for each block is extracted. Finally, a complete decision table for the program is created by merging the tables from each block (**Figure 5**). This approach reduces the number of paths extracted. For example, with the conventional method, the number of paths extracted in the case of subroutine calls equals the product of the number of subroutine levels while with our technique, the number is equal to the sum of the number of subroutine levels. This reduction increases with the degree of subroutine call nesting. For example, for a subroutine call that has been nested three times, the conventional technique extracts 3,060 paths while our technique extracts 41.

Use of decision tables will shorten the time it takes to analyze a system and improve the accuracy of the results. Furthermore, BRMS rules can be effectively extracted by translating the decision tables into BRMS rule sets.

6. Conclusion

We introduced an approach to program transformation that extracts partial components from the programs in a system and determines the best transformation solution for each one in accordance with its

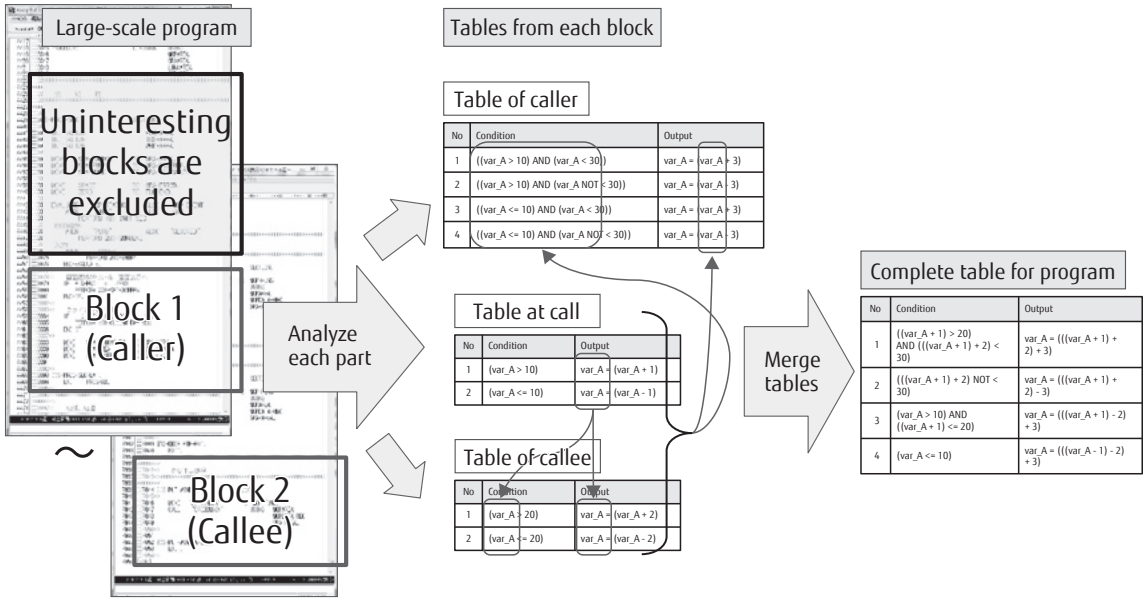

Figure 5
Merging tables created for each block.

characteristics. We also introduced three analysis techniques to support system modernization. The software map technique is provided as an application management service by Fujitsu and is already in practical use.

In an internal trial of selecting programs with high values of business logic complexity, we demonstrated that the quantification of the business logic complexity agreed well with the results of manual analysis done by systems engineers. The technique for creating a decision table was found to require approximately 33% less time than the time required to generate specification documents. We also demonstrated that the tables can be accurately translated into BRMS rules.

The novel approach and three analysis techniques described in this paper enable the analysis of current systems, which must have high stability and reliability because systems of record (SoR) must be robust against change. The approach also enables planning for the transformation of individual functions extracted as partial components, such as transforming functions into a service or BRMS or continuing to use them as they are. Our system also enables cooperation with systems of engagement (SoE), which will become increasingly necessary for connecting customers and enterprises.

References

1) IPA/SEC: A user guide that leads to successful reconstruction of the system. *http://www.ipa.go.jp/sec/reports/20170131.html, in Japanese*.
2) K. Kobayashi et al.: Feature-Gathering Dependency-Based Software Clustering Using Dedication and Modularity. ICSM2012, pp. 462–471 (2012).
3) K. Kobayashi et al.: SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints. ICPC2013, pp. 43–52 (2013).
4) M. Kamimura et al.: Measuring Business Logic Complexity in Software Systems. APSEC2015, pp. 370–376 (2015).
5) Y. Maeda et al.: Testcase Generation based on Symbolic Execution for COBOL Applications. Foundation of Software Engineering XIX, pp. 196–200, 2012 (in Japanese).
6) Y. Maeda et al.: A Testcase Generation Method to Cover Branches for Business Applications. Foundation of Software Engineering XXII, pp. 65–70, 2015 (in Japanese).

Yoshiharu Maeda
Fujitsu Laboratories Ltd.
Mr. Maeda is currently engaged in the research of software testing and software analysis.

Manabu Kamimura
Fujitsu Laboratories Ltd.
Dr. Kamimura is currently engaged in the research of software analysis and software metrics.

Keisuke Yano
Fujitsu Laboratories Ltd.
Mr. Yano is currently engaged in the research of software analysis and software visualization.

