2014 6th International Workshop on Empirical Software Engineering in Practice

Extraction of Conditional Statements for Understanding Business Rules

Tomomi Hatano*, Takashi Ishio*, Joji Okada†, Yuji Sakata† and Katsuro Inoue*
*Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {t-hatano,ishio,inoue}@ist.osaka-u.ac.jp
†NTT DATA Corporation, Research and Development Headquarters, Center for Applied Software Engineering
Toyosu Center Building Annex, 3-9, Toyosu 3-chome, Koto-ku, Tokyo 135-8671, Japan
Email: {okadaju,sakatayu}@nttdata.co.jp

***Abstract*—In the maintenance of a business system, developers must understand the computational business rules implemented in the system. Computational business rules define how an output value of a feature is computed from inputs; the rules are represented by conditional statements in the source code. Unfortunately, understanding business rules is a tedious and error-prone activity. Since a feature computes various outputs, developers must analyze the implementation of the feature and extract the conditional statements relevant to a particular output. In this paper, we propose a program dependence analysis technique tailored for understanding business rules. Given a variable representing an output, our approach extracts conditional statements that may affect the computation of the output. To evaluate the usefulness of the approach, we conducted an experiment with eight developers in a company. The results showed that our approach enables developers to accurately identify conditional statements relevant to business rules.**

I. Introduction

In the maintenance of a business system, developers must understand computational business rules implemented in the system [1]. Computational business rules define how the output of a feature is computed from inputs. An output value is affected by conditional statements, *e.g.,* nested `if` statements in the feature source code. The rules are often described by a table listing all possible output values and their corresponding conditions.

Understanding computational business rules is a tedious and error-prone activity for two main reasons [2]. First, documentation describing the rules is generally lost, outdated, or otherwise unavailable. Second, a feature of a system may compute multiple outputs. For each output, developers are required to answer a question: which conditional statements are relevant to this output value? The developers must then analyze the implementation of the feature and extract the relevant conditional statements in order to understand the business rules

Backward program slicing [3] is used to understand business rules [2], [4], [5], [6], [7]. Cosentino *et al.* [2] propose an application of program slicing to extract statements relevant to business rules that compute a particular variable. However, they report that the extracted statements may include conditional statements that are irrelevant to the business rules. The irrelevant statements are called *technical statements* in [2] because they often check if system resources, such as a data file or a database connection, are available for executing a feature. The technical statements themselves do not affect the output directly, although they do decide whether the computation is executed. Program slicing extracts both types of conditional statements since it does not differentiate between the two. Sneed *et al.* [8] conclude that techniques for data flow analysis and for extracting partial paths are required for understanding business rules.

We propose a program dependence analysis technique tailored for understanding computational business rules. Given a variable representing an output, our approach extracts conditional statements that may affect the computation of the value of the variable. To exclude technical statements from the analysis, we use a partial control-flow graph, every path of which outputs a computed result. In addition, we ensure the specified variable is data dependent on a statement that is directly or transitively dependent on the extracted conditional statements.

We have evaluated whether this approach actually contributes to the performance of developers investigating business rules. The evaluation was a controlled experiment based on an actual process in a company. Eight subjects in the company were asked to identify conditional statements relevant to business rules for a system output. The results showed that our approach enables developers to more accurately identify conditional statements relevant to business rules, without affecting the time required for the task.

The contributions of the paper are summarized as follows.

- We propose a program dependence analysis technique for understanding business rules. Our approach is a variant of program slicing that excludes technical statements.
- We evaluate our technique by conducting an experiment involving eight industrial experts. To the best of our knowledge, this is the first study to apply an automated extraction technique to experts' tasks in business rule reverse engineering.

The rest of the paper is organized as follows. Section II provides a motivating example. Sections III and IV, respectively, describe and evaluate our approach. Section V lists related work. Section VI describes the conclusion and future work.

978-1-4799-6666-0/14 $31.00 © 2014 IEEE
DOI 10.1109/IWESEP.2014.14

TABLE I: Tables representing computational business rules for the fee and time limit

(a) fee

values	conditions
5	children
10	students
15	adults

(b) time limit

values	conditions
3	premium members
2	no members

II. MOTIVATING EXAMPLE

Throughout this paper, we use an example feature that includes the business rules of an imaginary facility. The feature computes a usage fee and a time limit for the facility. The charge is 15 dollars for adults, 10 dollars for students, and 5 dollars for children. The time limit is 2 h for regular members and 3 h for premium members. Tables I(a) and I(b) describe the computations for the fee and time limit, respectively. Children may not become premium members.

The feature is implemented by a single method in Figure 1. The method action takes two variables as input: status, representing a user type (child / student / other), and member, representing whether the user is a premium member. The method computes two output variables corresponding to a usage fee and a time limit. The output variables are represented by the parameters of the setFee and setHour methods.

The method action comprises three steps. The first step checks whether the database access at line 2 produced an error. The second step computes an output fee from lines 5 through 14, following the rules shown in Table I(a). Lines 7 through 9 examine a constraint between two input variables and cancel the computation if the constraint is violated. The third step computes an output hour at lines 15 through 18, following the rules shown in Table I(b).

Developers maintaining the system must recover Tables I(a) and I(b) from the source code in Figure 1 in order to understand the computational business rules of the feature. To recover the tables, developers must answer the question: *Which conditional statements are relevant to the values passed to* setFee *and* setHour*?*

Backward program slicing appears to be a promising tool to answer the question. A program slice is computed using a program dependence graph. This graph is a directed graph in which the vertices represent all executable statements in a program; the edges represent the control and data dependencies among the statements. These dependencies are computed using a control-flow graph. Figure 2a shows a control-flow graph of the method in Figure 1.

In the graph, each node has a label indicating its corresponding line number. In a control-flow graph, a statement s_2 is control dependent on a statement s_1, if s_1 determines whether s_2 is executed. For example, lines 5 and 6 are control dependent on line 2, because line 2 has another control-flow path that reaches the exit of the method without visiting lines 5 and 6. A statement s_2 is data dependent on a statement s_1, if s_2 may use a variable whose value is defined by s_1. For example, line 14 is data dependent on line 5 because the value assigned at line 5 is used at line 14, if the execution passes through lines 6, 11, and 14. Figure 2b shows the resultant

```
1  public void action(int status, boolean member) {
2    if (hasError()) { // irrelevant
3      return;
4    }
5    int fee = 15;
6    if (status == STAT_CHILD) { // relevant to setFee
7      if (member) { // irrelevant
8        return;
9      }
10     fee = 5;
11   } else if (status == STAT_STUDENT) { // relevant to setFee
12     fee = 10;
13   }
14   setFee(fee);
15   setHour(2);
16   if (member) { // relevant to setHour
17     setHour(3);
18   }
19   return;
20 }
```

Fig. 1: An example method implementing business rules

program dependence graph of the method. A backward slice with respect to a variable used in a statement is the set of vertices that are reachable from the vertex representing the statement by backward traversal via edges.

Although backward program slicing extracts all statements that may affect a given variable, it cannot answer the question of which statements are relevant to the given variable. For example, a backward program slice with respect to the variable fee at line 14 includes four conditional statements (lines 2, 6, 7, and 11) that may be executed before line 14. However, only lines 6 and 11 are relevant to the computational business rules for fee, since the variable is assigned by statements controlled by those statements. On the other hand, lines 2 and 7 are irrelevant to the rules in Table I(a) since they represent conditions to decide whether the feature is executed.

Similarly, we can extract statements that may affect a parameter passed to setHour at lines 15 and 17, by computing the union of program slices with respect to the lines (a decomposition slice [9]). However, the resultant slice includes four conditional statements at lines 2, 6, 7, and 16. The statement at line 16 is relevant to the computational business rules for the parameter, while the other statements are irrelevant. As demonstrated in these examples, program slicing does not distinguish conditional statements that are relevant to the business rules from other conditional statements. Consequently, developers must manually extract the conditional statements relevant to the computational business rules for the output.

III. OUR APPROACH

Our approach is a program dependence analysis of a single method in a Java program, where we analyze data dependencies caused by method calls in the method. Our approach takes two inputs: method m that implements the business rules to be analyzed and a setter method s called in m that receives the output of the business rules. Our approach extracts the conditional statements in m that are *relevant* to s. A conditional statement c is relevant to s, if c directly or transitively affects a statement that determines an argument for method s. Conditional statements which are not relevant to s

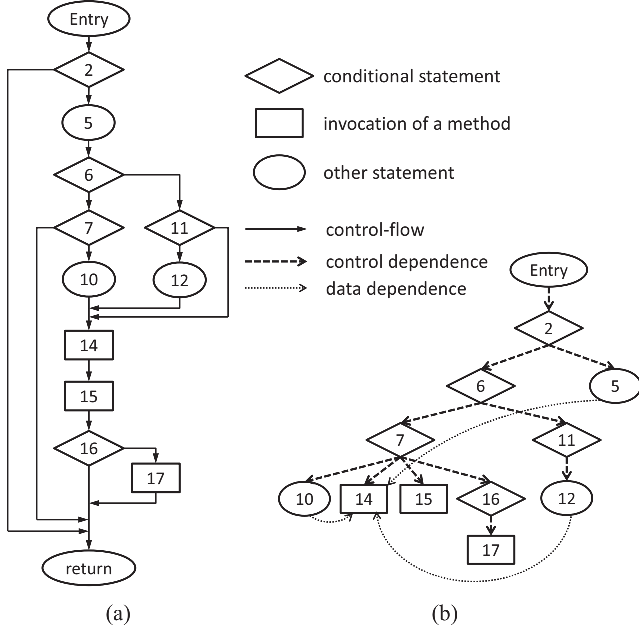

Fig. 2: A control-flow graph (a) and program dependence graph (b) of Figure 1

include technical statements and statements relevant to other setter methods.

Our approach comprises three steps.

1) Extract a control-flow graph (CFG) of the method m and its subgraph G_s related to s.
2) Extract control dependence edges in the CFG and G_s and data dependence edges in G_s.
3) Extract relevant conditional statements from method m using control-flow, control dependence, and data dependence edges.

Our approach uses a call graph for the entire program to identify method call instructions in m that invoke s and to perform data dependence analysis on method calls in m. We use variable-type analysis [10] for our implementation.

A. Control-Flow Analysis

This step constructs a CFG from the bytecode of m, and extracts its subgraph such that every path from the entry point invokes s. A CFG is a directed graph in which the vertices V_{CFG} represent all bytecode instructions of m and the edges CF represent control-flow paths [11]. Let S be the set of instructions invoking s. Subgraph G_s has vertices V_s and edges CF_s formulated as follows.

$$\begin{aligned} V_s &= \{v \in V_{CFG} \mid \exists s \in S : v \xrightarrow{CF*} s\} \\ CF_s &= \{(v1, v2) \in CF \mid v1 \in V_s \wedge v2 \in V_s\} \end{aligned}$$

$x \xrightarrow{E} y$ denotes there exists an edge from x to y in E (*i.e.*, $(x, y) \in E$). $x \xrightarrow{E*} y$ denotes there exists a path from x to y through edges in E. Note that $x \xrightarrow{E*} x$.

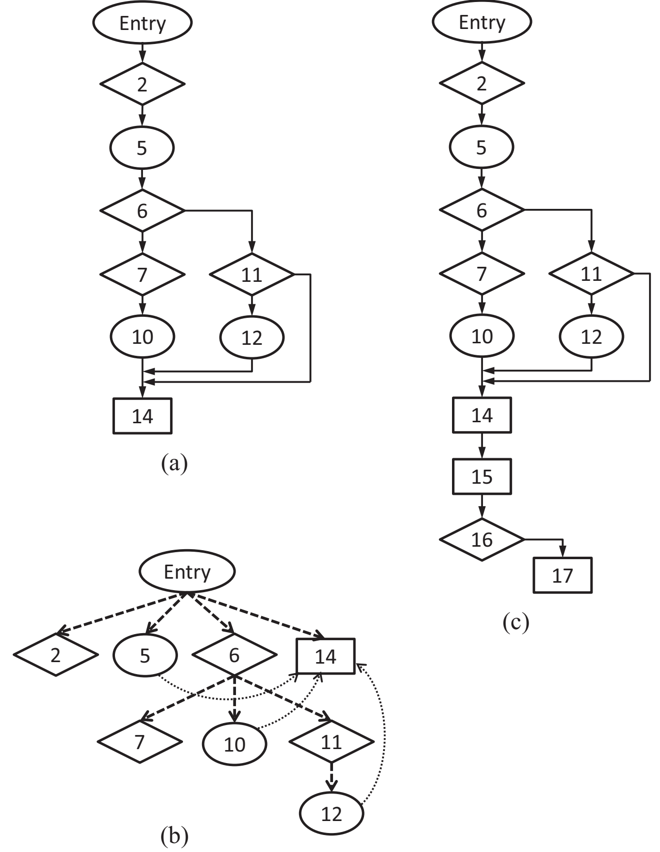

Fig. 3: Three graphs to extract conditional statements: (a) is a subgraph of Figure 2a for setFee (line 14). (b) is a dependence graph extracted from (a). (c) is a subgraph of Figure 2a for setHour (lines 15 and 17).

Figure 3a shows an example subgraph of the CFG in Figure 2a with respect to setFee. For simplicity, the vertices in Figure 3 represent executable statements and their line numbers in the program although actual vertices of our implementation represent bytecode instructions. While vertices 2 and 7 have branches in the CFG, they have no branches in the subgraph. Thus, the conditional statements corresponding to the vertices are not relevant to the computational business rules for fee in Section II.

B. Dependence Analysis

This step extracts data dependence edges (DD_s) and two kinds of control dependence edges (CD and CD_s). CD is the set of control dependence edges extracted from the CFG of m. CD_s and DD_s are sets of control dependence and data dependence edges extracted from the subgraph G_s. In addition to the definition of dependence representations given by Horwitz *et al.* [12], we extract the following data dependence edges.

1) Constant values: A constant value used in a statement is independent of other statements. However, we define a data dependence edge between a bytecode instruction that loads a constant value and another instruction that uses the value.

For example, the statement at line 17 comprises two bytecode instructions: the instruction that loads the constant value 3 and the instruction that invokes setHour. There exists a data dependence edge between the two. This data dependence is introduced to identify a conditional statement that controls method call statements using different constant values.

2) Field and array variables: Suppose an instruction i_1 defines the value of a field variable (or an element of an array variable) and another instruction i_2 uses the value of the field variable (or element of an array variable). There exists a data dependence edge from i_1 to i_2 if i_1 and i_2 may access the same field (or the same array). Each field is identified by class name and field name considering class hierarchy. Each array is identified by its type.

3) Invocations of methods: Suppose instructions i_1 and i_2 invoke methods. There exists a data dependence edge from i_1 to i_2 if the following condition holds.

$$Def(i_1) \cap Use(i_2) \neq \emptyset$$

where $Def(i_1)$ is the set of field and array variables that may be defined by methods (directly or transitively) invoked from the instruction i_1. $Use(i_2)$ is the set of field and array variables that may be used by methods (directly or transitively) invoked from i_2. For a conservative analysis, we assume that library methods that are not included in the target program may define and use all field and array variables in the program.

C. Extracting conditional statements

Using the computed dependence edges, this final step extracts the set of relevant conditional statements R from m as follows.

$$\begin{aligned} R &= CV \cup OW \\ CV &= \{c \mid \exists s \in S, \exists d \in V_s : c \xrightarrow{(CD_s \cup DD_s)*} d \xrightarrow{DD_s} s\} \\ OW &= \{c \mid \exists s_1, s_2 \in S, \exists d \in V_s : s_1 \xrightarrow{CF_s*} c \wedge \\ &\qquad c \xrightarrow{(CD \cup DD_s)*} d \xrightarrow{DD_s} s_2\} \end{aligned}$$

CV represents the set of conditional statements that may affect statements passing values to s. Each element of CV directly or transitively affects an instruction that provides data to s. OW represents the set of conditional statements that decide whether a value set by s_1 is overwritten by another value at s_2. Since a conditional statement affects an output even if it decides not to execute s_2, we use CD instead of CD_s for the definition of OW.

Figure 3b shows a dependence graph of the program in Figure 1 when setFee is specified as s (*i.e.,* $S = \{14\}$). The conditional statements at lines 6 and 11 are extracted as relevant statements since they hold the condition of CV. Figure 3c shows a subgraph when setHour is specified as s (*i.e.,* $S = \{15, 17\}$). The conditional statement at line 16 is extracted as a relevant statement since it holds the condition of OW. The conditional statements at lines 2 and 7 are not extracted since they do not hold the conditions of either CV or OW; nor do they satisfy the condition of CV since they have no dependence edge to other vertices. Furthermore, they do not satisfy the condition of OW since they are not reachable from setFee or setHour.

R may include truly irrelevant statements since our approach uses only dependencies among instructions. If several assignment statements pass the same value to s, conditional statements that select one of those statements are irrelevant to the output. However, our approach regards such conditional statements as relevant to the output.

Our implementation supports two techniques for providing the extracted conditional statements to developers. The first one is code comments. Our tool adds code comments to conditional statements as shown in Figure 1. Since developers are required to analyze the same method m for each output variable, an irrelevant statement for one variable may be relevant for another. Developers may use code comments generated for several variables to understand the entire structure of the method. The second technique is a CSV file. Our tool outputs a file listing all the conditional statements in a specified method m and indicating whether each statement is relevant or not. Developers may record the progress of investigation in the generated file.

IV. Evaluation

Developers must examine the source code of a feature to understand the computational business rules, even if the relevant conditional statements are extracted by our technique. In order to evaluate whether our technique can help developers identify relevant conditional statements, we conducted a controlled experiment using human subjects. We formulated the following research questions.

RQ1 *Can our technique help developers accurately identify conditional statements relevant to computational business rules?*

RQ2 *Can our technique reduce the time needed to identify relevant conditional statements?*

RQ3 *To what degree do the results of our technique differ from those of the developers' investigation?*

A. Experimental Setup

1) Subjects: We recruited eight reverse engineering experts from a company. They had been engaged in reverse engineering for at least a year. Their Java experience was widely distributed from 0.5 to 12 years, with a median of a year. No subject was familiar with the target system.

2) Tasks: The tasks used in our experiment were created from **MosP 4.0.0**[1], an attendance management system. Two methods, m_1 and m_2, were randomly selected from the longest methods whose conditional statements could not be removed by program slicing. Table II shows the details of the two methods. Column $|C|$ represents the number of all conditional statements in m. Column $|C_s|$ represents the number of conditional statements located prior to a setter method call s in the source code of m. All conditional statements were extracted by program slicing with respect to each setter method. Column $|R|$ indicates the number of conditional statements extracted by our technique.

For each task, the subjects were given the following.

- Eclipse IDE including the source code of the system.

[1] http://sourceforge.jp/projects/mosp/releases/53354

TABLE II: Target methods

ID	Methods	LOC	$\|C\|$	$\|C_s\|$	$\|R\|$
T1	m_1 = getPaidHolidayDataDto s_1 = setAcquisitionDate	101	17	12	7
T2	m_2 = chkWorkOnHolidayInfo s_2 = setPltWorkType	152	23	23	15

TABLE III: Task assignment

	Task 1		Task 2	
Subject	Target	Our approach	Target	Our approach
1, 2	T1	Yes	T2	No
3, 4	T1	No	T2	Yes
5, 6	T2	Yes	T1	No
7, 8	T2	No	T1	Yes

- Target method m and the setter method s to be analyzed.
- Spreadsheet including all conditional statements and their line numbers in m.

The subjects performed one task with our technique and the other task without our technique. Table III shows the tasks assigned to the subjects. The results of our technique were provided to the subjects by annotating conditional statements in the source code (as shown in Figure 1) and in a spreadsheet. A subject working without our technique received a list of conditional statements in a spreadsheet without annotation. A program slice was not explicitly provided because it includes all the conditional statements located prior to s.

Each task comprised two subtasks that are typical reverse engineering processes in the company. In the first subtask, the subjects classified each conditional statement as either *relevant* or *irrelevant* and wrote the result into a given spreadsheet. In the second subtask, they used the results of the first subtask to create a table of the computational business rules. Each task is limited to 2 h. The results of the second subtask were used to determine the correct answer of the first subtask.

3) Procedure: At the beginning of the experiment, the subjects were given the following information: (1) the purpose of the experiment, (2) a summary of our technique, (3) the process of the task, (4) an exercise in **MosP** using an example task, and (5) an explanation of the answer of the example task. The subjects performed their tasks independently after the introduction.

After all the tasks had been completed, the subjects discussed the correct answer with the third author, who is also a reverse engineering expert in the company. Since they reached agreement on the computational business rules in the tasks, we used the results to evaluate the accuracy of the subjects.

B. Results

1) RQ1 Can our technique help developers accurately identify conditional statements relevant to computational business rules?: The left box plot in Figure 4 compares the accuracy of the developers' classification of conditional statements. The accuracy is the ratio of the number of correctly classified conditional statements against the total number of conditional statements in the method. We observed that developers supported by our technique classified conditional statements more accurately. A Wilcoxon rank sum test showed the difference was statistically significant (the p-value was 0.0148). Furthermore, Cohen's d = 0.846 showed the difference was large [13]. The improvement was achieved because subjects without our technique tend to accidentally misclassify conditional statements as irrelevant. Our technique enabled subjects to carefully investigate such relevant conditional statements by identifying irrelevant conditional statements. We concluded that our approach enabled the developers to accurately identify conditional statements relevant to computational business rules.

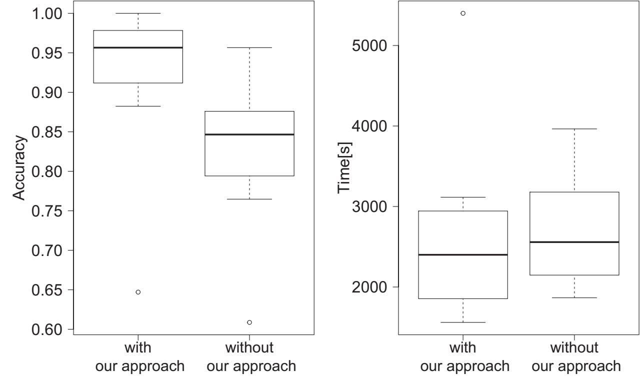

Fig. 4: Comparison of the accuracy and time for tasks

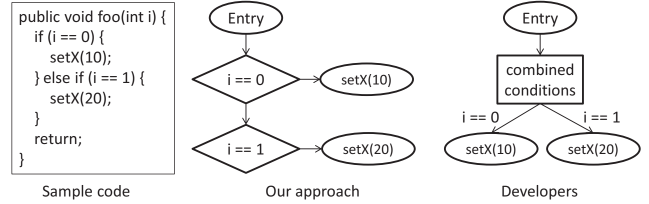

Fig. 5: The difference between our approach and developers

2) RQ2 Can our technique reduce the time needed to identify relevant conditional statements?: The right box plot in Figure 4 compares the time spent to complete the task with and without our technique. Although developers supported by our technique took less time than those without our technique, the difference was not statistically significant (the p-value was 0.6454). This is because the subjects read the entire source code of the methods to understand business rules. Even if relevant conditional statements are automatically extracted, they must verify what conditions are represented in those statements. We conclude that our technique does not affect the time for investigating source code and creating tables.

3) RQ3 To what degree do the results of our technique differ from those of the developers' investigation?: The set of relevant conditional statements created during the discussion

with the subjects included 17 statements. Fourteen of the original 22 statements were extracted by our technique and the remaining three conditional statements were missed by our technique. Hence, the recall and the precision of our technique are 0.82 (14/17) and 0.64 (14/22), respectively. Our technique included eight statements that were classified as irrelevant by the subjects, because of a simple conservative analysis for library methods. The conditional statements would be excluded if a more precise analysis was implemented.

Our technique missed three conditional statements because of a difference between actual dependence and conceptual dependence. A simplified example is shown in Figure 5. In the source code, two conditional statements, `if (i == 0)` and `if (i == 1)`, determine a value passed to the method `setX`. Our approach classified the former statement as *relevant* and the latter statement as *irrelevant*, because the former statement determined the parameter: 10 is passed if `i == 0` and 20 otherwise. On the other hand, developers classified both conditional statements as relevant since they subconsciously regarded the two consecutive statements as a single control-flow structure.

We determined that our technique can extract conditional statements without missing relevant statements by regarding consecutive conditional statements as a combined statement as shown in the right side of Figure 5. Although conditional statements extracted by this approach may include irrelevant statements, the approach is expected to reduce the developers' identification time since they only need to consider the extracted statements without inspecting the other conditional statements.

V. Related work

Various techniques to extract computations from source code have been proposed. The frameworks to extract business rules are proposed [2], [4], [5], [6]. They represent computations of business variables using program slicing techniques. Our analysis technique excludes irrelevant statements that are extracted by these techniques. Moreover, we have evaluated the ability of our technique to assist developers.

Pichler [14] proposed a symbolic execution technique to extract computations from a Fortran program. Their technique requires actual test cases from a target system, while our approach does not assume the existence of test cases. Dubinsky *et al.* [15] propose a method to identify business rules in the code and find the code locations. The method relies on business terms in the code and comments while we use only dependencies among statements. Thin slicing [16] extracts only assignment statements that define an output value and excludes all conditional statements from a slice. Our technique extracts conditional statements that depend on assignment statements. Decomposition slicing [9] extracts all statements that may affect the assignment statements of a particular variable. Since a decomposition slice is computed by the union of traditional program slices, it includes conditional statements that are irrelevant to business rules as described in our motivating example. As a variant of program slicing, our approach is related to amorphous slicing [17]. Amorphous slicing summarizes the computations of a particular variable while retaining the semantics of the original program. Our approach extracts conditional statements relevant to the computations of a particular variable while ignoring irrelevant control-flow paths; therefore, it does not preserve the semantics.

VI. Conclusion and future work

We have proposed a program dependence analysis technique tailored for understanding computational business rules. Our approach extracted conditional statements that were relevant to an output value. We conducted a controlled experiment to evaluate whether or not the approach actually contributed to the performance of the developers. We found that our approach enabled developers to more accurately identify conditional statements relevant to computational business rules.

In future work, we would like to support conceptually related conditional statements as described in Section IV-B3. We are also interested in the interprocedural analysis of business rules scattered across several methods. Finally, we plan to apply our approach to other enterprise systems to evaluate the effectiveness of our approach.

Acknowledgment

We would like to thank the subjects who participated in this study. This work was supported by JSPS KAKENHI Nos.25220003 and 26280021.

References

[1] K. Wiegers and J. Beatty, *Software Requirements*, 3rd ed. Miscrosoft press, 2013.

[2] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from COBOL: A model-based framework," in *Proc. WCRE*, 2013, pp. 409–416.

[3] M. Weiser, "Program Slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, 1984.

[4] X. Wang, J. Sun, and X. Yang, "Business rules extraction from large legacy systems," in *Proc. CSMR*, 2004, pp. 249–253.

[5] H. Huang and W. Tsai, "Business rule extraction from legacy code," in *Proc. COMPSAC*, 1996, pp. 162–167.

[6] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application," in *Proc. RuleML*, 2012, pp. 17–31.

[7] H. Sneed, "Extracting business logic from existing COBOL programs as a basis for redevelopment," in *Proc. IWPC*, 2001, pp. 167–175.

[8] H. Sneed and K. Erdos, "Extracting business rules from source code," in *Proc. WPC*. IEEE Comput. Soc. Press, 1996, pp. 240–247.

[9] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 751–761, 1991.

[10] V. Sundaresan and L. Hendren, "Practical virtual method call resolution for Java," in *Proc. OOPSLA*, 2000, pp. 264–280.

[11] F. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[12] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM TOPLAS*, vol. 11, no. 3, pp. 345–387, 1989.

[13] J. Cohen, "Statistical power analysis," *Current Directions in Psychological Science*, vol. 1, no. 3, pp. 98–101, 1992.

[14] J. Pichler, "Specification extraction by symbolic execution," in *Proc. WCRE*, 2013, pp. 462–466.

[15] Y. Dubinsky, Y. Feldman, and M. Goldstein, "Where is the business logic?" in *Proc. ESEC/FSE*, 2013, pp. 667–670.

[16] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proc. PLDI*, 2007, pp. 112–122.

[17] M. Harman, D. Binkley, and S. Danicic, "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, no. 1, pp. 45–64, Oct. 2003.