IEICE TRANSACTIONS

on Information and Systems

VOL. E99-D NO. 4
APRIL 2016

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Dependency-Based Extraction of Conditional Statements for Understanding Business Rules

Tomomi HATANO[†a)], ***Nonmember***, **Takashi ISHIO**[†b)], ***Member***, **Joji OKADA**[††c)], **Yuji SAKATA**[††d)], ***Nonmembers***, ***and*** **Katsuro INOUE**[†e)], ***Fellow***

SUMMARY For the maintenance of a business system, developers must understand the business rules implemented in the system. One type of business rules defines computational business rules; they represent how an output value of a feature is computed from the valid inputs. Unfortunately, understanding business rules is a tedious and error-prone activity. We propose a program-dependence analysis technique tailored to understanding computational business rules. Given a variable representing an output, the proposed technique extracts the conditional statements that may affect the computation of the output. To evaluate the usefulness of the technique, we conducted an experiment with eight developers in one company. The results confirm that the proposed technique enables developers to accurately identify conditional statements corresponding to computational business rules. Furthermore, we compare the number of conditional statements extracted by the proposed technique and program slicing. We conclude that the proposed technique, in general, is more effective than program slicing.

key words: *static analysis, control-flow analysis, data-dependence analysis, reverse engineering, Java*

1. Introduction

For the maintenance of a business system, developers must understand the business rules implemented in the system [1]–[3]. Business rules are classified into several types, such as computational business rules and constraints [1]. Computational business rules define how the output of a feature is computed from the valid inputs. Constraints restrict the actions that the system or its users are allowed to perform. In the implementation of these rules, conditional statements (*e.g.,* `if` statements) affect output values in the computations and verify that the constraints are not violated.

Understanding business rules is a tedious and error-prone activity for two main reasons [4]. First, the documentation describing the rules is typically lost, outdated, or otherwise unavailable. Second, the implemention of the rules is scatterd throughout the source code. Developers are required to extract conditional statements corresponding to the business rules. When understanding computational business rules, developers must answer which of the conditional statements correspond to the computational business rules for each output of a feature.

Backward program slicing [5] is used to understand business rules [2]–[4], [6], [7]. Cosentino *et al.* [4] proposed an application of program slicing to extract statements corresponding to business rules that compute a particular variable. However, they reported that the extracted statements may include conditional statements that do not correspond to the business rules. Those statements are called *technical statements* [4] because they frequently verify whether system resources, such as a data file or database connection, are available for executing a feature. The technical statements themselves do not affect the output directly, although they do determine if the computation is executed. Furthermore, the extraction based on program slicing does not distinguish the types of rules, although Wiegers *et al.* state that distinguishing them is helpful to understand business rules. Consequently, program slicing is not enough to understand business rules because it may extract technical statements and does not distinguish the types of rules.

We propose a program-dependence analysis technique tailored to understanding computational business rules. Given a variable representing an output, the proposed technique extracts the conditional statements that may affect the computation of the value of the variable. To exclude technical statements from the analysis, we construct a partial control-flow graph (CFG), every path of which outputs a computed result. Further, we ensure that the specified variable is data-dependent on a statement that is directly or transitively dependent on the extracted conditional statements. Our technique is designed to extract conditional statements corresponding to computational business rules, whereas the existing techniques extract multiple types of rules. In this paper, conditional statements corresponding to computational business rules are called *relevant statements*.

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

Manuscript received May 25, 2015.
Manuscript revised October 17, 2015.
Manuscript publicized January 8, 2016.

[†]The authors are with Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The authors are with Research and Development Headquarters, Center for Applied Software Engineering, NTT DATA Corporation, Tokyo, 135–8671 Japan.
a) E-mail: t-hatano@ist.osaka-u.ac.jp
b) E-mail: ishio@ist.osaka-u.ac.jp
c) E-mail: okadaju@nttdata.co.jp
d) E-mail: sakatayu@nttdata.co.jp
e) E-mail: inoue@ist.osaka-u.ac.jp
DOI: 10.1587/transinf.2015EDP7202

The contributions of this paper are summarized as follows.

- We propose a program-dependence analysis technique for understanding business rules. The proposed technique 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.
- We apply the proposed technique and program slicing to two systems developed in industry and demonstrate that the proposed technique extracts a reduced number of conditional statements.

Note that this paper is an extension of our earlier work [8]. We conduct a comparison with program slicing in this paper.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 provides a motivating example. Sections 4 and 5, respectively, describe and evaluate the proposed technique. Section 6 offers our conclusions and future work.

2. Related Work

Sneed *et al.* [9] proposed a framework based on a program slicing technique to extract business rules from source code. They concluded that techniques for data flow analysis and extracting partial paths are required for understanding business rules. The framework is extended for COBOL [6], C/C++ [2], Java [7], respectively. Furthermore, Cosentino *et al.* [4] extended the framework for COBOL programs based on the principles of Model Driven Engineering. They automatically identify variables representing outputs using the COBOL command and visualize the extracted rules at a higher abstraction level. Although they do not evaluate their technique, a preliminary experiment indicates that their extraction based on program slicing includes conditional statements that do not correspond to business rules. Our technique enables the exclusion of those statements and extracts conditional statements corresponding to computational business rules, while the existing techniques extract multiple types of rules without distinguishing them. Furthermore, we conducted a controlled experiment to evaluate the ability of the proposed technique to help developers.

Various variants of program slicing have been proposed for different situations. Thin slicing [10] extracts only assignment statements that define the value of a given variable. It excludes all conditional statements from a program slice for code inspection and debugging of large-scale systems. Decomposition slicing [11] extracts statements that may affect all the statements using a given variable. A decomposition slice is computed from the union of traditional program slices to capture all computations on the variable for software maintenance. Amorphous slicing [12] transforms statements extracted by program slicing to simplify a program while preserving the semantics of the program. The simplification (*e.g.,* expanding function calls) is convenient in the context of program comprehension. Our technique focuses on partial control-flow paths for a given variable to understand computational business rules.

Dubinsky *et al.* [13] proposed a method to identify business rules in the code using information retrieval techniques. They found that the quality of their technique depended on terms used in identifiers and comments. Because idiosyncratic abbreviations are frequently used in the code of a business system, developers require knowledge of the system. Our dependency-based technique is independent of identifiers and comments.

Pichler [14] proposed a symbolic execution technique to extract computations from Fortran programs. Symbolic execution enables the computations to be represented by equations. However, it typically has low scalability owing to the fact that all the paths of a program must be analyzed (called path explosion problem). Furthermore, loop statements and invocations of libraries are challenging for symbolic execution. To overcome these challenges, their technique requires actual test cases and their execution results. Jaffar *et al.* [15] proposed a path-sensitive control-flow graph where a statement may be represented by multiple vertices. This graph is constructed by symbolic execution and slicing the results (called tree slicing). Although they attempt to reduce the number of the paths to be analyzed by merging vertices, the path explosion problem is a challenge for them. Because our technique excludes conditional statements that do not correspond computational business rules, it reduces the number of the paths compared to traditional slicing. We expect that our technique can contribute to the path explosion problem in the extraction of business rules.

3. Motivating Example

3.1 Business Rules Implemented in Source Code

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 for adults, $10 for students, and $5 for children. The time limit is 2 hours for regular members and 3 hours for premium members. Tables 1(a) and 1(b) describe the computational business rules for the fee and time limit, respectively. The facility defines a constraint; children cannot become premium members.

The feature is implemented by the single method in

Table 1 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

```
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

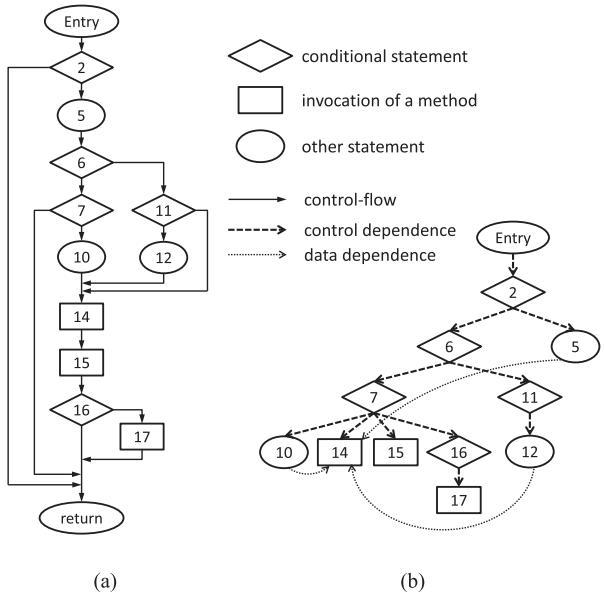

Fig. 2 A control-flow graph (a) and program dependence graph (b) of Fig. 1

Fig. 1. The method action requires two variables as input: status, representing a user type (child / student / other), and member (regular / premium). 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 includes three steps. The first step verifies if the database access at line 2 produced an error. The second step computes an output fee from lines 5 through 14, following the rules presented in Table 1(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 presented in Table 1(b).

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

3.2 Extraction of Business Rules by Program Slicing

Backward program slicing [5] appears to be a promising technique to respond to the above question. The technique extracts all the statements that may affect the value of a given variable, referred to as the *slicing criterion*. The set of the extracted statements is called the program slice.

A program slice is computed using a program dependence graph. This graph is a directed graph where the vertices represent all the executable statements in a program; the edges represent the control and data dependencies among the statements. A statement s_2 is control dependent on a statement s_1, if s_1 determines whether s_2 is executed. 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. These dependencies are extracted from a CFG. This graph is a directed graph where the vertices represent all executable statements (or basic block) in a program; the edges represent the control-flow paths [16].

Figures 2(a) and 2(b) illustrate examples of a CFG and program dependence graph. In the graphs, each vertex has a label indicating the corresponding line number. Each graph has a special vertex named Entry that represents the entry of a method and controls statements that are not control-dependent on any statements.

A program slice with respect to a slicing criterion is extracted by backward traversal of a program dependence graph. The traversal visits all vertices that are reachable from the vertex representing the criterion via edges. A set of the visited vertices and criterion is the program slice for the criterion. For example, given line 14 as a slicing criterion, program slicing extracts lines {2, 5, 6, 7, 10, 11, 12, 14}.

Although backward program slicing extracts all statements that may affect the value of a given variable, it cannot answer the question of which of the statements are relevant to the given variable. For example, a 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 correspond to the computational business rules for fee, because the value of fee is defined by the user type (see Table 1(a)). Lines 2 and 7 do not correspond to the computational rules for fee because line 2 is a technical statement which verifies database connections and line 7 is a condition for the constraint to children. They only determine if the feature is executed.

When investigating the computational business rules for the time limit, 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 [11]). However, the resultant slice includes four conditional statements at lines 2, 6, 7, and 16 ,

whereas only line 16 corresponds to the computational business rules for the time limit. Lines 2 and 7 do not correspond to the computational business rules for the same reasons of the former example. Whereas line 6 is relevant to setFee, a value of status does not affect a value of a parameter passed to setHour on the paths that execute setHour. For setHour, line 6 is a condition representing the constraint; it is not a part of computational business rules.

As demonstrated in these examples, program slicing does not distinguish conditional statements corresponding to the computational business rules from other conditional statements. Consequently, developers must manually extract the conditional statements corresponding to the computational business rules for the output.

4. The Proposed Technique

The proposed technique 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. The proposed technique requires two inputs: a 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. The proposed technique 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 that are not relevant to s include technical statements and statements relevant to other setter methods.

The proposed technique includes three steps:

1. Extract a 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.

The proposed technique 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 [17] for our implementation.

4.1 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 where the vertices V_{CFG} represent all the bytecode instructions of m and the edges CF represent control-flow paths [16]. Let S be the set of instructions invoking s. Subgraph G_s has vertices V_s and edges CF_s formulated as follows.

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

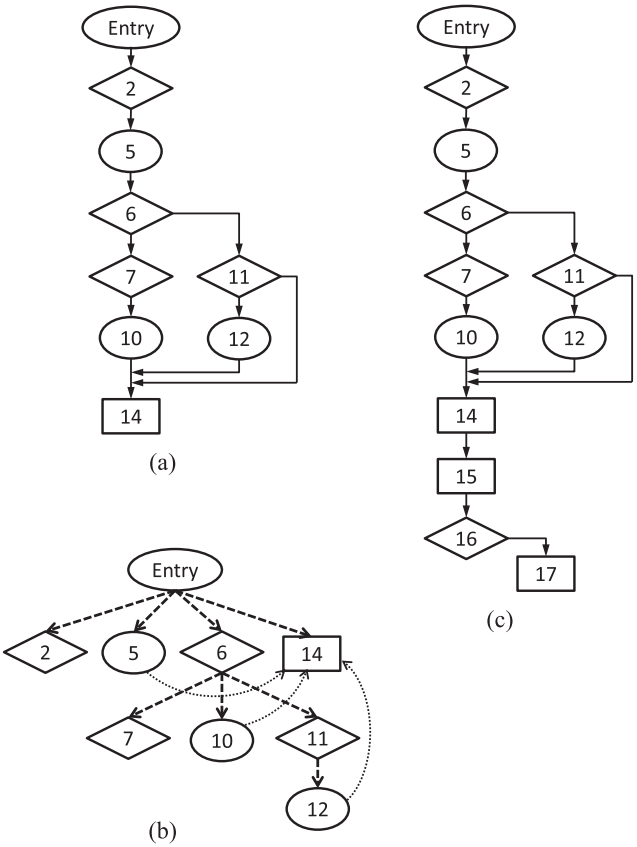

Fig. 3 Three graphs to extract conditional statements: (a) is a subgraph of Fig. 2(a) for setFee (line 14). (b) is a dependence graph extracted from (a). (c) is a subgraph of Fig. 2(a) for setHour (lines 15 and 17).

$x \xrightarrow{E} y$ 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 $x \xrightarrow{E*} x$.

Figure 3(a) is a sample subgraph of the CFG in Fig. 2(a) with respect to setFee. For simplicity, the vertices in Fig. 3 represent executable statements and their line numbers in the program although actual vertices of our implementation represent bytecode instructions. Whereas 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 Sect. 3.

4.2 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.* [18], we extract the following data-dependence edges.

4.2.1 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 includes 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.

4.2.2 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 a field variable (or an 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.

4.2.3 Invocations of Methods

The side effect of method calls is conservatively analyzed to avoid overlooking relevant statements. 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.

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

$Def(i_1)$ 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.

For example, suppose that setFee in the Fig. 1 defines a value of a field A.x and setHour uses a value of the same field A.x, data-dependence edges from an invocation of setFee to invocations of setHour are extracted.

4.3 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 determine if a value set by s_1 is overwritten by another value at s_2. Because 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 3(b) presents the dependence graph of the program in Fig. 1 when setFee is specified as s (*i.e.,* $S = \{14\}$). The conditional statements at lines 6 and 11 are extracted as relevant statements because they hold the condition of CV. Figure 3(c) displays 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 because it holds the condition of OW. The conditional statements at lines 2 and 7 are not extracted because 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 because they are not reachable from setFee or setHour.

R may include truly irrelevant statements because the proposed technique uses only dependencies among instructions. If several assignment statements pass the same value to s, conditional statements that select one of these statements are irrelevant to the output. However, the proposed technique 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 indicated in Fig. 1. Because 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 can use the 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. Developers can record the progress of the investigation in the generated file.

5. 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 the proposed technique. To evaluate whether the proposed technique can help developers identify relevant conditional statements, we conducted a controlled experiment using human subjects. Our research questions are formulated as follows:

RQ1 *Does the proposed technique help developers accurately identify conditional statements relevant to computational business rules?*

RQ2 *Does the proposed technique affect the time required to identify relevant conditional statements?*

Table 2 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

RQ3 *Is the proposed technique accurate?*

Because the controlled experiment investigates a particular case, it is not obvious that the proposed technique, in general, is more effective than program slicing. We compared the proposed technique with program slicing to answer another research question formulated as follows:

RQ4 *How many conditional statements are removed from the program slices?*

We applied our technique and program slicing to all the methods that implement business rules in two subject systems written in Java and compared the number of conditional statements extracted by the techniques.

5.1 Experiment with Human Subjects

5.1.1 Setup

(1) Subjects

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

(2) Tasks

The tasks used in our experiment were created from **MosP 4.0.0**[†], an attendance management system. Two Java methods, m_1 and m_2, were randomly selected from the longest methods whose conditional statements could not be removed by program slicing. Table 2 presents 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 extracted by program slicing with respect to each setter method (s_1 and s_2). All the conditional statements in C_s are located prior to each setter method. Column $|R|$ indicates the number of conditional statements extracted by the proposed technique.

For each task, the subjects were given the following:

- Eclipse IDE including the source code of the system.
- 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 the proposed technique and the other task without the proposed technique.

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

Table 3 Task assignment

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

Table 3 indicates the tasks assigned to the subjects. The results of the proposed technique were provided to the subjects by annotating conditional statements in the source code (as illustrated in Fig. 1) and in a spreadsheet. A subject working without the proposed 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 included 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 recorded the result in 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 was limited to two hours. 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 the proposed technique, (3) the process of the task, (4) an exercise in MosP using a sample task, and (5) an explanation of the answer for the sample task. The subjects performed their tasks independently after the introduction.

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

5.1.2 Results

RQ1: Does the proposed technique help developers accurately identify conditional statements relevant to computational business rules?

The left box plot in Fig. 4 compares the accuracy of the developers' classification of conditional statements. The accuracy is the ratio of the number of correctly classified conditional statements to the total number of conditional statements in the method. We observed that developers supported by the proposed technique classified the conditional statements more accurately. A Wilcoxon rank sum test indicated that the difference was statistically significant (the p-value was 0.0148). Furthermore, Cliff's Delta [19], which measures the effect size for the test, indicated that the difference was large (the delta was 0.625) [20]. The improve-

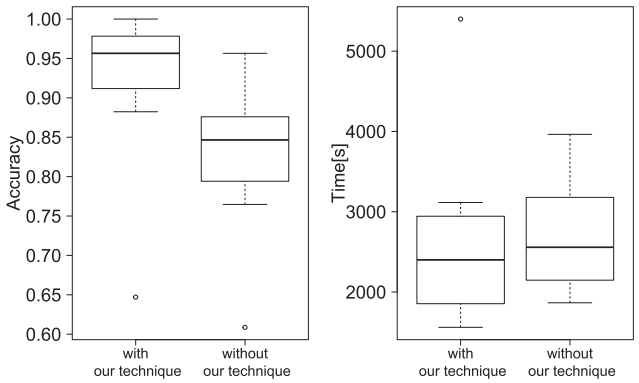

Fig. 4 Comparison of the accuracy and time for tasks

ment was achieved because the subjects without the proposed technique tended to accidentally misclassify conditional statements as irrelevant. The proposed technique enabled subjects to carefully investigate such relevant conditional statements by identifying irrelevant conditional statements. We concluded that the proposed technique enabled the developers to accurately identify conditional statements relevant to computational business rules.

RQ2: Does the proposed technique affect the time required to identify relevant conditional statements?

The right box plot in Fig. 4 compares the time required to complete the task with and without the proposed technique. Although developers supported by the proposed technique required less time than those without the proposed technique, the difference was small and not statistically significant (the delta was −0.172 and the p-value was 0.645). This is because the subjects read the entire source code for the methods to understand the business rules. Even if relevant conditional statements are automatically extracted, they must verify what conditions are represented in those statements. We conclude that the proposed technique does not affect the time for investigating source code and creating tables.

RQ3: Is the proposed technique accurate?

It is our opinion that the proposed technique accurately extracts relevant statements though it sometimes includes irrelevant statements and misses relevant statements. There were 17 relevant conditional statements created during the discussion with the subjects. Fourteen of the original 22 statements were extracted by the proposed technique and the remaining three conditional statements were missed by the proposed technique. Hence, the recall and the precision of the proposed technique are 0.82 (14/17) and 0.64 (14/22), respectively. The proposed 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.

The proposed technique missed three conditional statements because of a difference between actual dependence

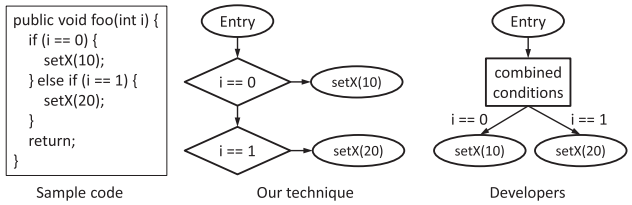

Fig. 5 The difference between our technique and developers

and conceptual dependence. A simplified example is illustrated in Fig. 5. In the source code, two conditional statements, `if (i == 0)` and `if (i == 1)`, determine a value passed to the method `setX`. The proposed technique 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. Conversely, developers classified both conditional statements as relevant because they subconsciously regarded the two consecutive statements as a single control-flow structure.

We determined that the proposed technique can extract conditional statements without missing relevant statements by regarding consecutive conditional statements as a combined statement as indicated in the right side of Fig. 5. Although conditional statements extracted by this technique may include irrelevant statements, the technique is expected to reduce the developers' identification time because they are only required to consider the extracted statements without inspecting the other conditional statements.

5.2 Comparison with Program Slicing

5.2.1 Setup

We extracted conditional statements from all the methods that implement business rules in two systems: MosP and a small sales management system, which is used in a company for a system development exercise. Using naming rules of class and method, we identified methods M that implement business rules and setter methods S that receive the outputs. M and S in MosP are identified as follows:

- ***M*:** all the methods that belong to classes ending with "Action"
- ***S*:** all the methods that start with "set" and belong to classes ending with "Vo" or "Dto"

In MosP, Action classes have methods that implement business rules. The methods store the computational results into DTO objects to transfer the results to a database. Further, the methods store the computational results into VO objects to display the results on the user interface. We identified M and S in the sales management system in a similar manner. In this experiment, we analyzed all the pairs of $m \in M$ and $s \in S$ that were directly invoked by m.

Table 4 The extraction results of conditional statements

System		MosP	Sales
#targets		1,440	28
#(our < slice)		991	28
median	our	1	1
	slice	2	8.5
	all	4	10
max	our	48	4
	slice	65	19
	all	70	19
sum	our	4,381	20
	slice	7,527	224
	all	11,831	248

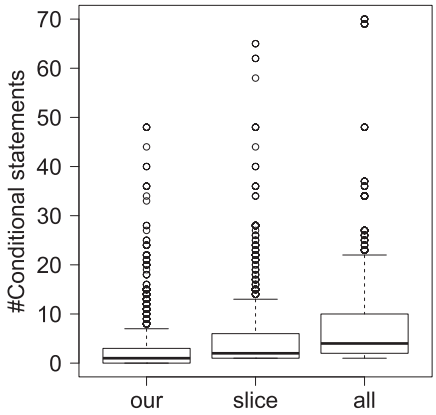

Fig. 6 The number of conditional statements in MosP

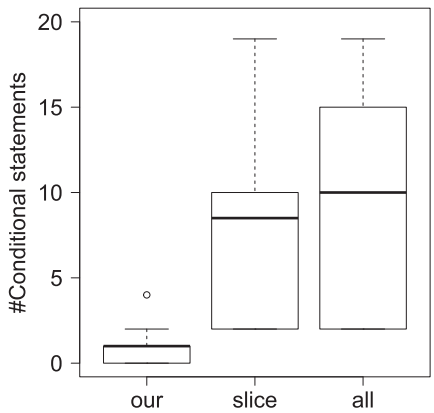

Fig. 7 The number of conditional statements in the sales management system

5.2.2 Result

RQ4: How many conditional statements are removed from the program slices?

Table 4 presents the extraction results of conditional statements. Row #targets represents the number of method pairs where the number of conditional statements extracted by program slicing is larger than zero. Row #(our < slice) represents the number of method pairs where the number of conditional statements extracted by the proposed technique is smaller than that of the conditional statements extracted by program slicing. The remainder of Table 4 represents the statistics of the number of conditional statements. Figures 6 and 7 plot the distributions of the number of conditional statements in MosP and the sales management system, respectively.

In MosP, for 69% (991/1,440) method pairs, the number of conditional statements extracted by the proposed technique was smaller than that of the conditional statements extracted by program slicing. A test using a R package[†] estimated that the number of conditional statements extracted by the proposed technique was 1.75 smaller with the median than that of conditional statements extracted by program slicing. We consider that the reduction is effective for developers investigating the computational business rules because they must analyze all possible method pairs in the system. From the sum indicated in Table 4, we conclude that the proposed technique can reduce conditional statements that developers must analyze to 58% (4,381/7,527) of program slicing.

In the sales management system, for all method pairs, the number of conditional statements extracted by the proposed technique was smaller than that of the conditional statements extracted by program slicing. Furthermore, the reduction size was greater than that in MosP (the estimated median was 5.75). This is because the computational business rules were simple, whereas the violation checks for inputs were large and complicated. The proposed technique excluded conditional statements for the checks, whereas program slicing extracted them.

[†]http://cran.r-project.org/web/packages/exactRankTests/

5.3 Threats to Validity

In the controlled experiment with human subjects, we used the discussion results of the nine experts as the correct answer. These results may be wrong because the experts were not developers of the subject system. Moreover, the results may be biased because they may have wanted to believe their own answer. However, we believe that this possibility is low because the nine experts did finally agree on the same answer.

Because the controlled experiment was conducted on a single case, different results may be observed on other companies. However, we consider that the evaluation follows an actual situation in understanding business rules because the subject system is developed by a different company from the one that the participants work for. Furthermore, we open the answer used in the evaluation on our website [††] to make the experiment replicable.

In the comparative experiment with program slicing, we used the naming rules of classes and methods to identify the methods to be analyzed. The first author reviewed the source code of the subject systems and determined that

[††]http://sel.ist.osaka-u.ac.jp/people/t-hatano/ieice/exp.html

using the naming rules was valid. However, the analyzed methods may include inappropriate methods and there may exist other methods that should be analyzed. We did not read all the methods in the subject systems.

We obtained the comparison results from two systems. The results may not be applicable to arbitrary business systems. However, we believe that the proposed technique can be effective in general business systems because the two systems had different uses (attendance and sales management) and were developed by different organizations.

6. Conclusion and Future Work

We have proposed a program-dependence analysis technique designed for understanding computational business rules. The proposed technique extracts conditional statements that are relevant to an output value. We conducted a controlled experiment to evaluate whether this technique actually contributed to the performance of developers. We determined that the proposed technique enabled developers to more accurately identify conditional statements relevant to computational business rules. Furthermore, we compared the number of conditional statements extracted by the proposed technique and program slicing. We confirmed that the proposed technique is more effective for developers investigating computational business rules compared to program slicing.

In future work, we would like to support conceptually related conditional statements as described in the result of RQ3. We are also interested in the interprocedural analysis of business rules distributed across several methods. Finally, we plan to apply the proposed technique to other enterprise systems to evaluate the effectiveness of the proposed technique.

Acknowledgments

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, Third ed., Microsoft Press, 2013.
[2] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Business rules extraction from large legacy systems," Proc. CSMR, pp.249–253, 2004.
[3] H. Sneed, "Extracting business logic from existing COBOL programs as a basis for redevelopment," Proc. IWPC, pp.167–175, 2001.
[4] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from COBOL: A model-based framework," Proc. WCRE, pp.409–416, 2013.
[5] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol.SE-10, no.4, pp.352–357, 1984.
[6] H. Huang, W. Tsai, S. Bhattacharya, X.P. Chen, Y. Wang, and J. Sun, "Business rule extraction from legacy code," Proc. COMPSAC, pp.162–167, 1996.
[7] 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," Proc. RuleML, pp.17–31, 2012.
[8] T. Hatano, T. Ishio, J. Okada, Y. Sakata, and K. Inoue, "Extraction of conditional statements for understanding business rules," Proc. IWESEP, pp.25–30, 2014.
[9] H. Sneed and K. Erdos, "Extracting business rules from source code," Proc. WPC, pp.240–247, IEEE Comput. Soc. Press, 1996.
[10] M. Sridharan, S.J. Fink, and R. Bodik, "Thin slicing," Proc. PLDI, pp.112–122, 2007.
[11] 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.
[12] M. Harman, D. Binkley, and S. Danicic, "Amorphous program slicing," Journal of Systems and Software, vol.68, no.1, pp.45–64, 2003.
[13] Y. Dubinsky, Y. Feldman, and M. Goldstein, "Where is the business logic?," Proc. ESEC/FSE, pp.667–670, 2013.
[14] J. Pichler, "Specification extraction by symbolic execution," Proc. WCRE, pp.462–466, 2013.
[15] J. Jaffar and V. Murali, "A path-sensitively sliced control flow graph," Proc. ESEC/FSE, pp.133–143, 2014.
[16] F. Allen, "Control flow analysis," ACM Sigplan Notices, vol.5, no.7, pp.1–19, 1970.
[17] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for Java," Proc. OOPSLA, pp.264–280, 2000.
[18] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," ACM TOPLAS, vol.11, no.3, pp.345–387, 1989.
[19] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," Psychological Bulletin, vol.114, no.3, pp.494–509, 1993.
[20] J. Romano, D. Kromrey, Jeffrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?," Proc. FAIR, pp.1–33, 2006.

Tomomi Hatano received his master's degree from Osaka University in 2015. He is a Ph.D. candidate at Osaka University. His research interests include program analysis and reverse engineering.

Takashi Ishio received the Ph.D. degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006-2007. He is now an assistant professor of computer science at Osaka University. His research interests include program analysis and program comprehension. He is a member of the IEICE, IPSJ, JSSST, IEEE, and ACM.

Joji Okada received master's degree in information science from Nagoya University in 2008. He is an assistant manager at NTT DATA Corporation. His interests are program analysis and programming. He is a member of the IPSJ.

Yuji Sakata received master's degree in materials science and engineering from the University of Tokyo in 1996. He is a manager at NTT DATA Corporation. His interests are program analysis and reverse engineering. He is a member of the IPSJ.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984-1986. He was a research associate at Osaka University from 1984-1989, an assistant professor from 1989-1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.