Tracking Data Dependence of Large-scale Systems for Practical Program Understanding

Submitted to
Graduate School of Information Science and Technology
Osaka University

July 2017

Tomomi HATANO

Abstract

Software developers must understand program behavior through code reading for software maintenance. When changing existing features, developers must understand locations of the source code related to the features. When fixing bugs, developers must understand which statements cause the bugs.

Existing studies reported that developers spend a lot of time for program understanding. Developers are often required to read the source code of large-scale systems that are unfamiliar to them. Therefore, many researchers investigated how developers understand programs and developed numerous techniques to help develepers understand programs.

Program dependence analysis is one of techniques for helping program understanding. It analyzes the source code to extract read/write relationships of variables (called data dependence), method call relationships, and so on. Developers use these relationships to explore the source code effectively and efficiently.

This dissertation describes studies on dependence analysis techniques for program understanding. These studies aim to extract useful information from the source code and provide it for developers. Furthermore, we released our analysis tool to facilitate future studies on dependence analysis by other researchers.

First, we conducted an empirical study to statistically investigate the effectiveness of *thin slicing*, which is a variant of program slicing to extract statements that produce data used by a particular statement. Although an existing study showed that thin slicing is useful for program understanding in small cases, it is not clear whether thin slicing is effective for program understanding in general. We computed thin slices with respect to all statements that consume data and measured various metrics on extracted statements. The results showed that the size of the extracted statements is small enough on average. Furthermore, we found that 10% of thin slices can be effective for identifying the source statements of data. We believe that these slices help developers track data dependence.

Second, we developed a novel dependence analysis technique tailored to understanding how outputs of a feature are computed from inputs (called *business rules*). Existing techniques extract statements that correspond to business rules.

However, these techniques may include conditional statements that do not correspond to rules. Our technique excludes those conditional statements by constructing a partial control-flow graph, every path of which outputs a computed result. We evaluated whether this technique actually contributes to the performance of developers who extract business rules. A controlled experiment based on an actual understanding process in one company shows that the technique enables developers to more accurately identify business rules without affecting the time required for the task. This is the first study to apply an automated extraction technique to practical tasks in business-rule understanding.

Finally, we developed a program analysis tool for Java named SOBA. It analyzes intra-procedural control-flow, data dependence, control dependence, method call relationships, and so on. Its design enables to easily obtain the above information without detailed knowledge of program analysis. We compared the functional differences and usage differences between SOBA and existing tools. We also compared the performance of them and showed that SOBA was faster than existing tools. Furthermore, SOBA was applicable to a large-scale system which has over 67,000 classes. We believe that releasing the analysis tool as open source software contributes to future studies of software engineering.

List of Publications

Major Publications

1. Tomomi Hatano, Yu Kashima, Takashi Ishio, Katsuro Inoue. “A Statistical Evaluation of Thin Slice Size”, In Proceedings of SES2013, pp1–6, 2013 (in Japanese).

2. Tomomi Hatano, Yu Kashima, Takashi Ishio, Katsuro Inoue. “A Statistical Evaluation of Thin Slice Size”, IPSJ Journal, Vol.55, No.2, pp.971–980, 2014 (in Japanese).

3. Tomomi Hatano, Takashi Ishio, Joji Okada, Yuji Sakata, Katsuro Inoue. “Dependency-Based Extraction of Conditional Statements for Understanding Business Rules”, IEICE Transactions on Information and Systems, Vol.E99-D, No.4, pp.1117–1126, 2016.

4. Tomomi Hatano, Takashi Ishio, Katsuro Inoue. “SOBA: A Simple Tookit for Java Bytecode Analysis”, Computer Software, Vol.33, No.4, pp.4–15, 2016 (in Japanese).

5. Tomomi Hatano, Takashi Ishio, Joji Okada, Yuji Sakata, Katsuro Inoue. “Extraction of Conditional Statements for Understanding Business Rules”, In Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice, pp25–30, Osaka, Japan, 2014.

Related Publications

1. Tomomi Hatano, Takashi Ishio, Yoshitaka Moro, Yuji Sakata, Katsuro Inoue. “Understanding Business System through Software Clustering Using I/O Instructions for External Systems”, In Proceedings of SES2015, pp17–27, Yokohama, 2015 (in Japanese).

2. Tomomi Hatano, Matsuo Akihiko. “Removing Code Clones from Industrial Systems Using Compiler Directives”, In Proceedings of the 25th International Conference on Program Comprehension, 336–345, Buenos Aires, Argentina, 2017.

Acknowledgement

I am deeply grateful to my supervisor, Professor Katsuro Inoue. Thanks to his continual support and valuable comments, I accomplished this study. I am extremely happy and blessed to have the opportunity of his supervision. I am convinced that all the things I have learned will be my wealth throughout my life.

I would like to express my gratitude to Professor Toshimitsu Masuzawa and Professor Shinji Kusumoto for helpful comments and suggestions on this dissertation. I would also like to acknowledge the guidance of Professors Yasushi Yagi.

I am indebted to Associate Professor Takashi Ishio at Nara Institute of Science and Technology. Throughout my works, he gave me a great deal of comments on conducting research, writing papers, presentation, and so on. His continual support greatly contributed to this dissertation.

I would like to thank to Associate Professor Makoto Matsushita and Assistant Professors Kula Raula Gaikovina at Nara Institute of Science and Technology. They gave me helpful comments and suggestions for my research presentation.

I wish to thank to Dr. Yu Kashima and all the members of Inoue Laboratory. I could have productive days thanks to their kind support. I would also like to acknowledge the support of the team members at my company.

I would like to express the appreciation to Mr. Joji Okada and Mr. Yuji Sakata at NTT DATA Corporation. They powerfully helped our development of a new technique and an experiment.

Finally, I would like to thank to my family. They have always respected my decision and supported my school life.

Contents

1 Introduction **1**

1.1 Program Understanding 1
1.2 Understanding Developers' Understanding 2
1.2.1 Eye tracking 2
1.2.2 Biometrics 2
1.3 Helping Developers' Understanding 3
1.3.1 Natural language processing 3
1.3.2 Software metrics 3
1.3.3 Software clustering 4
1.3.4 Execution trace 4
1.3.5 Static program dependence analysis 5
1.4 Challenges of Program Dependence Analysis for Understanding . 5
1.5 Contributions of the Dissertation 6
1.5.1 Evaluation of an existing analysis technique through an empirical study 6
1.5.2 Application of a dependence analysis technique to a practical understanding task 6
1.5.3 Development of a program analysis tool 7
1.6 Overview of the Dissertation 7

2 An Empirical Study on Thin Slice Size **9**

2.1 Introduction 9
2.2 Related Work 10
2.2.1 Program slicing 10
2.2.2 Reducing slice size 11
2.2.3 Thin slicing 12
2.2.4 Statistics on Slice Size 14
2.3 Research Question 14
2.4 Implementation of Thin Slicing 15

2.4.1 Intra-procedural data dependence analysis . . . 15
2.4.2 Inter-procedural data dependence analysis . . . 16
2.4.3 Data dependence analysis on heap . . . 16
2.4.4 Directed graph of bytecode instructions . . . 16
2.5 Experiment . . . 16
2.5.1 Measuring metrics . . . 16
2.5.2 Subject programs . . . 19
2.5.3 Results . . . 20
2.6 Conclusion of This Chapter . . . 21

3 Extracting Conditional Statements for Understanding Business Rules 23
3.1 Introduction . . . 23
3.2 Related Work . . . 25
3.3 Motivating Example . . . 26
3.3.1 Business rules implemented in source code . . . 26
3.3.2 Extraction of business rules by program slicing . . . 27
3.4 The Proposed Technique . . . 28
3.4.1 Control-flow analysis . . . 30
3.4.2 Dependence analysis . . . 31
3.4.3 Extracting conditional statements . . . 32
3.5 Evaluation . . . 33
3.6 Experiment with Human Subjects . . . 34
3.6.1 Setup . . . 34
3.6.2 Results . . . 36
3.7 Comparison with Program Slicing . . . 38
3.7.1 Setup . . . 38
3.7.2 Results . . . 39
3.8 Threats to Validity . . . 40
3.9 Conclusion of This Chapter . . . 41

4 Development of Program Analysis Tool for Java 43
4.1 Introduction . . . 43
4.2 Existing Tools and Our Motivation . . . 44
4.3 SOBA . . . 45
4.3.1 Characteristics of SOBA . . . 46
4.3.2 Example program . . . 48
4.3.3 Example studies using SOBA . . . 50
4.4 Comparison with Soot and WALA . . . 51
4.4.1 Programming . . . 51
4.4.2 Features . . . 52

4.4.3 Performance 53
4.5 Conclusion of This Chapter 54

5 Conclusion **55**
5.1 Summary of Studies 55
5.2 Future Directions 56

List of Figures

2.1 An example program. 11
2.2 A system dependence graph for computing slices. 12
2.3 A dependence graph for computing thin slices. 14
2.4 Bytecode representation corresponding to the program of Figure 2.1. 17
2.5 A dependence graph of bytecode instructions. 18
2.6 Cumulative frequency distribution for $|Backward(v)|$ 22

3.1 An example method implementing business rules 27
3.2 A control-flow graph (a) and program dependence graph (b) of Figure 3.1 29
3.3 Three graphs to extract conditional statements: (a) is a subgraph of Figure 3.2(a)for setFee (line 14). (b) is a dependence graph extracted from (a). (c) is a subgraph of Figure 3.2(a)for setHour (lines 15 and 17). 30
3.4 Comparison of the accuracy and time for tasks 36
3.5 The difference between our technique and developers 37
3.6 The number of conditional statements in MosP 39
3.7 The number of conditional statements in the sales management system 40

4.1 An example program which analyzes method call relationships using Soot 45
4.2 An example program which analyzes method call relationships using WALA 46
4.3 A class diagram of SOBA. 47
4.4 An example program which analyzes method call relationships using SOBA. 49
4.5 Execution results of Figure 4.4. 50
4.6 An example program which analyzes data dependence using SOBA. 50
4.7 Execution results of Figure 4.6. 51

List of Tables

2.1 A list of bytecode instructions and their types. 15
2.2 Subject programs. 20
2.3 Summary of metrics 21
2.4 Ratio of thin slices whose $|Method(Backward(v))| \geq 2$ and $|Source(v)| \leq 3$. 22

3.1 Tables representing computational business rules for the fee and time limit 26
3.2 Target methods 35
3.3 Task assignment 35
3.4 The extraction results of conditional statements 39

4.1 Classes of SOBA. 48
4.2 Comparison of the line number, class number, and command line options 51
4.3 Comparison of features and their usage of each tool. 52
4.4 Measured objects for performance comparison 53
4.5 Performance comparison of programs analyzing call relationships (time[ms] and memory[MB]) 53
4.6 Performance comparison of programs analyzing data and control dependence (time[ms] and memory[MB]) 54
4.7 Performance for Eclipse 4.2 and JDK 1.7.0 (67,973 classes and 543,425 methods) 54

Chapter 1

Introduction

Software systems play an important role in the modern society. They help to make people's lives more prosperous. Furthermore, many organizations rely on software systems for their operations.

To meet continuously changing requirements of software users, developers must perform maintenance tasks for existing software systems correctly and quickly. Outdated systems are not worth for users because their requirements change in a short time. In addition, systems that do not work properly can affect negative impacts on their users.

1.1 Program Understanding

Software developers must understand existing programs for maintaining software systems. Program understanding is a process of understanding features, structures, and behavior of programs [1, 2]. Singer et al. [3] reported that developers perform understanding tasks before changing source code because developers must explore source code relevant to the intended change. Mäder et al. [4] reported that developers who know source files related to features can produce a software change more efficiently [4].

Developers spend a lot of time with reading source code for understanding [1,5, 6]. Existing software systems are getting larger and more complex because developers have changed them for many years to meet users' requirements. Furthermore, developers often read unfamiliar code written by someones they do not know.

1.2 Understanding Developers' Understanding

Many studies have been conducted to reveal how developers understand programs and what information is important for understanding. To answer these questions, researchers observed developers who perform understanding tasks using various methods. The results from their studies tell us how we should support developers who understand programs.

1.2.1 Eye tracking

Eye tracking is useful for exploring how developers read the source code. Crosby [7] et al. used eye tracking to investigate how developers read the source code. They explored the visual attention of developers who read a binary search algorithm written in Pascal. The result showed that developers need numerous fixations in most areas of the source code. Sharif [8] et al. investigated the effect of identifier-naming conventions (i.e., camelCase and under_score) on program understanding. They used an eye tracker to capture quantitative data and replicate a previous study where Binkley et al. [9] conducted an experiment to determine which identifier style is faster and more accurate for software maintenance. The result showed that developers recognized identifiers in the underscore style more quickly while there was no difference in accuracy between the two styles. Furthermore, Sharif et al. [10] investigated the relationship between scan time and defect detection time in source code review. They showed that the longer reviewers spend in the initial scan, the quicker they find the defect.

1.2.2 Biometrics

Some researchers utilize biometrics to reveal developers' cognitive process in program understanding. Parnin [11] analyzed electromyogram (EMG) signals of developers who perform programming tasks. EMG measures electrical signals emitted from muscle nerves. His experiment would suggest that EMG can measure the difficulty of understanding tasks. Siegmund et al. [12] applied functional magnetic resonance imaging (fMRI), which measures blood-oxygenation levels that change as a result of localized brain activity, to an experiment of program understanding. In the experiment, to reveal which brain regions are activated during the tasks, participants performed understanding tasks of the source code which implements standard algorithms (e.g., sorting) in the fMRI scanner. The results showed a clear activation pattern of five brain regions, which are related to working memory, attention, and language processing. Their study provided an empirical evidence that language processing is essential for program understanding.

1.3 Helping Developers' Understanding

Researchers have proposed many techniques and conducted experiments to help developers understand programs. The proposed techniques cover a wide range of research topics: the automatic generation of software document, software metrics, architecture recovering by software clustering, analysis of execution trace, and static program dependence analysis. This dissertation describes studies on static program dependence analysis.

1.3.1 Natural language processing

Some studies automatically generate English documentation from the source code using natural language processing techniques. Sridhara et al. developed summarization techniques to generate natural language comments for Java methods [13] and their formal parameters [14]. The techniques identify important statements to extract keywords from identifier names in those statements. A natural language processing technique stitches the keywords into English sentences. These sentences help developers understand the behavior a Java method. McBurney et al. [15] developed a summarization technique which includes the context of method invocations to explain why the method exists or what role it plays in the system. They conducted an experiment involving 12 participants to compare their summaries with Sridhara's summaries. The result showed that their summaries are superior in quality. Moreno et al. [16] developed a technique to generate summaries of Java classes. Their summaries allow developers to understand the main goal and structure of the class. Their experiment showed that most of their summaries are readable and understandable and they do not include extraneous information.

1.3.2 Software metrics

Some researchers use software metrics to identify which programs are difficult to understand. Katzmarski et al. [17] asked developers to order multiple programs by their complexity. The authors measured metrics of the programs to compare rankings by developers and metrics values. They found that a simple metrics which counts assignment statements was similar to developers' opinion. Kasto et al. [18] investigated the correlation between metrics and the difficulty of understanding tasks using student subjects. The students were asked to answer questions. The authors used the percentage of correct answers as the difficulty. The result showed that several metrics (such as cyclomatic complexity and nested block depth) were significantly correlated to the difficulty. The authors concluded that metrics can be useful tool in the early prediction of the difficulty. Singh et al. [19] used devel-

opers' activity logs, such as viewing a file, editing source code, as indicators of a understanding effort. The authors investigated the correlation the activity logs and software metrics. They showed that class cohesion, which measures relative number of directly connected methods in the class, had a significant correlation with the logs.

1.3.3 Software clustering

Software clustering is useful for understanding the overview and architecture of systems [20]. It decomposes a system into smaller manageable subsystems by measuring the similarity among software entities (e.g., files and functions).

Mancoridis et al. [21] developed a clustering technique which creates subsystems that have high-cohesion and low-coupling to other subsystems. The technique reduces the system complexity to help developers understand a system structure. Kobayashi et al. visualized a system structure [22] using their clustering technique [23]. The technique gathers classes that implement relevant features into a cluster. They extract method call relationships to identify relevant classes. Anquetil et al. [24] developed a clustering technique which uses the naming convention of files. Scanniello et al. [25] proposed a technique for understanding three-tier architecture of client-server systems. It extracts inheritance relationships among classes and interfaces of Java. It also extracts identifier names and measures their similarity. Tzerpos et al. [26] developed a rule-based clustering technique for C systems. In this technique, developers define rules such as gathering fileA.c and fileA.h into the same cluster, gathering device drivers into the same cluster, and gathering utility functions into the same cluster.

1.3.4 Execution trace

The analysis of execution traces helps developers understand the runtime behavior of programs. De Pauw et al. [27] developed a technique to visualize the behavior and an architecture of object-oriented systems. They showed that the technique is effective in their daily work for understanding large and complicated systems. Lange et al. [28] visualized program's interactions to examine a large amount of execution traces. Their visualization uses a graph which can present voluminous information effectively and allow developers to find helpful information. Reiss [29] developed a Java visualizer which describes a program action with low overhead while the program is running. The visualizer enables developers to understand what the program is actually doing. Greevy [30] introduced 3D visualization technique which helps developers understand overview of the dynamic behavior of features. Beck et al. [31] proposed an approach which displays the consumed run-

time for each method on the source code editor. The approach is useful for finding and fixing performance bottlenecks.

1.3.5 Static program dependence analysis

Program-dependence analysis techniques extract the following information from the source code to provide it for developers who understand programs.

Control-flow: execution paths including the orders and branches of statements.

Data dependence: read/write relationships of variables.

Control dependence: an evaluation result of a statement determines whether another statement is executed.

Method call relationships: a method calls another method.

Integrated development environments such as Eclipse often extract the above to visualize the dependence.

Program slicing [32] is one of dependence analysis techniques for helping developers perform understanding tasks [33]. It analyzes data and control dependence to extract all statements that may affect a statement specified by developers. It enables developers to investigate how the values of variables are computed [34]. Therefore, developers can understand why programs output wrong values in debugging [35]. Furthermore, program slicing techniques are embedded in a code inspection tool [36].

1.4 Challenges of Program Dependence Analysis for Understanding

Program slicing techniques may be ineffective for understanding large-scale systems because these techniques extract so many dependencies that developers cannot grasp. Binkley et al. [37] reported that program slicing extracts 30% of statements on average. From this result, it is difficult for developers to read the extracted statements for understanding large-scale systems.

While many researchers have proposed variants of program slicing that reduce information provided for developers by focusing on limited dependencies [38–44], thin slicing is one of promising techniques which track only data dependence between statements. It extracts statements that produce data used by a statement specified by developers. It is expected that thin slicing is effective for understanding large-scale systems because tracking data dependence is an important task for

understanding [45]. However, it is not clear whether thin slicing is effective for understanding in general though an existing study demonstrated its effectiveness in small cases.

It is difficult to apply existing dependence analysis techniques including program slicing to a practical understanding process. One of understanding tasks for industrial systems is answering a question: how outputs of a system are computed from inputs? (called *business rules*). For example, a facility defines a calculation procedure for admission fees that are 500 yen per child, 1,000 yen per student, and 1,500 yen per adult; this procedure is a business rule. Although existing studies have proposed various techniques to extract statements corresponding to business rules [46–50], the extracted statements may include irrelevant statements to business rules.

Finally, an easy-to-use dependence analysis tool is required for developing a novel analysis technique. Although existing tools are available for dependence analysis, their users must pay high learning cost to use them efficiently. Furthermore, they have too many features and options for researchers in software engineering who require basic information as described in Section 1.3.5 to conduct their studies.

1.5 Contributions of the Dissertation

This dissertation describes three studies to meet the above challenges.

1.5.1 Evaluation of an existing analysis technique through an empirical study

We conducted an empirical study to evaluate the effectiveness of thin slicing. Although an existing study showed that thin slicing is useful for program understanding in some cases [51], it is not clear whether thin slicing is effective in general. We measured various metrics on thin slices to investigate its effectiveness. The results showed that the average size of thin slices is small enough and 10% of thin slices can be effective for tracking data dependence. We believe that thin slicing can support developers to understand large-scale systems by visualizing data dependence.

1.5.2 Application of a dependence analysis technique to a practical understanding task

We developed a novel dependence analysis technique for understanding business rules and evaluated its effectiveness. Although understanding the rules is important

for maintaining business systems, existing techniques are not enough to help developers understand rules. The existing techniques are program slicing techniques that extract statements relevant to the rules. However, they may include statements that do not correspond to the rules. We developed a technique to exclude those statements by constructing control-flow graphs, every path of which is reachable to a given point. We evaluated whether our technique actually contributes to the performance of developers investigating the rules. A controlled experiment showed that our technique enables developers to more accurately identify statements corresponding to the rules without affecting the time. The task in our experiment is based on a practical understanding process which is actually performed in one company. This is the first study to apply the extraction technique to a practical understanding task of business rules and evaluate its effectiveness with human subjects.

1.5.3 Development of a program analysis tool

We developed a program analysis tool for Java. Java is a widely-used programming language in research and the real world. Although existing tools have enough features to perform program analysis, using them is difficult for those who do not have detailed knowledge of many analysis algorithms. While researchers in software engineering often require basic information about programs, they are not interested in detailed behavior and implementation of algorithms. Our tool, named SOBA, enables to easily obtain basic information such as control-flow, data dependence, control dependence, and method call relationships. We released SOBA as open source software to facilitate future studies on dependence analysis by other researchers. We also hope that SOBA will be the chance to learn algorithms and research on dependence analysis.

1.6 Overview of the Dissertation

Chapter 2 presents an empirical study on effectiveness of thin slicing. Chapter 3 explains our dependence analysis technique for understanding business rules. Chapter 4 describes the design and implementation of our analysis tool for future studies. Finally, Chapter 5 concludes this dissertation and discusses future directions.

Chapter 2

An Empirical Study on Thin Slice Size

2.1 Introduction

In software maintenance, developers spend a lot of time for program understanding [1, 6]. It is an important understanding task that locating the current implementation of a feature in the source code. Mäder et al. [4] reported that developers who know source files related to features can produce a software change more efficiently. To locate the source code related to features, developers need to explore data dependence among methods. Identifying multiple methods that use same data through parameters and fields is useful to understanding the source code locations [45]. However, it takes a lot of time for developers to explore data dependence [52]

Program slicing is a promising technique to reduce the time required for understanding [32]. It extracts all statements (called a slice) that may affect a particular statement. Although it reduces the source code lines developers must read, it extracts 30% of statements on average [37]. Therefore, it is difficult to use program slicing for understanding large-scale systems.

Thin slicing [51] is a variant of program slicing which reduces the slice size. It extracts only statements that produce data used by a particular statement. The size of thin slices is much smaller than that of traditional slices. Thin slicing can be effective for identifying locations where the same data is used because it focuses on only data-flow paths. It is expected that thin slicing is effective for the task to locate source code related to features. Sridharan et al. demonstrated 22 cases where thin slicing reduced the time required for program understanding [51].

Although thin slicing is a promising technique, it is not revealed whether thin

slicing is effective for program understanding in general. Existing studies measured the average size of slices to evaluate the effectiveness of slicing techniques. Binkley et al. [37] reported the average size of traditional slices and Jász et al [53] also reported the average size of the approximation of traditional slices. As these studies, the average size of thin slices must be measured to evaluate their effectiveness.

We implemented a thin slicer for Java programs to measure thin slice size. We address the question whether thin slicing is effective in general for program understanding in terms of the average slice size, while Sridharan et al. demonstrated only 22 cases where thin slicing is useful. If thin slices are small enough but capture inter-procedural data-flow paths, they would enable developers to efficiently investigate data-flow paths in the source code. On the other hand, if thin slices are contained in a single method, they would be less effective since developers can investigate data-flow paths by simply reading the source code. Thus, we also measure metrics about thin slices (such as the number of methods included in the slice and the number of producer instructions) to investigate how many thin slices can be effective for understanding.

2.2 Related Work

2.2.1 Program slicing

Program slicing is a technique which extracts all statements that may affect a particular statement [32]. A slice is computed using System Dependence Graph [54] (SDG) which is extended from Program Dependence Graph (PDG). PDG is a directed graph whose vertices representing each statement in a procedure of a program and edges representing dependencies among the vertices. SDG is a directed graph that connects PDGs of each procedure by procedure call relationship.

A slice with respect to a variable used in a statement, called a slicing criterion, is computed as a set of vertices that are reachable from the vertex representing the slicing criterion via edges in SDG. A backward slice and a forward slice can be obtained by backward and forward traversal, respectively. A backward slice is a set of statements that may affect a slicing criterion. A forward slice is a set of statements that may be affected a slicing criterion.

Figures 2.1 and 2.2 show an example program and its SDG. The vertices a and b represent formal parameters and the vertex ret represents a return value. A slice with respect to Line 11 is a set of Lines {3, 4, 5, 7, 8, 9, 10, 11, 16}. The slice tells that Lines 6 and 13 do not affect Line 11.

Program slicing is effective for program understanding and debugging. When developers understand source code fragments they focus on, program slicing en-

```
 1: public class Sample {
 2:   public static void m1() {
 3:     A x = new A();
 4:     A z = x;
 5:     int y = 1;
 6:     int i = 0;
 7:     A w = x;
 8:     w.f = add(y, 2);
 9:     if (w == z) {
10:       int v = z.f;
11:       System.out.println(v);
12:     }
13:     System.out.println(i);
14:   }
15:   public static int add(int a, int b) {
16:     return a + b;
17:   }
18: }
```

Figure 2.1: An example program.

ables developers to investigate how the values of variables are computed [34]. In debugging, program slicing is used to reveal why programs output wrong values [35]. Kusumoto et al. [55] reported that program slicing enables developers to effectively perform debugging tasks. Anderson et al. [36] embedded program slicing techniques in their code inspection tool.

2.2.2 Reducing slice size

Program slicing enables developers to read only source code which is relevant to a particular variable. A smaller slice is more effective to reduce the time for source code reading. Therefore, various methods have been proposed to reduce slice size. Chopping [38] computes intersection of a forward slice with respect to a vertex producing data and a backward slice with respect to a vertex using data. A chop indicates data-flow paths between two variables. While chopping is originally defined for intra-procedural slices, Reps et al. [39] extended chopping to inter-procedural slicing. Distance-limited slicing [40] is another extension of program slicing; it extracts vertices whose distance from slicing criteria is less than a given threshold. Chen et al. [41] proposed an interactive method for feature location based on program slicing. This method firstly visualizes only vertices which are connected to a slicing criterion. When a developer selected an interesting vertex, vertices connected to the selected vertex become visible. This method enables developers to explore a small part of a dependence graph. Callstack-sensitive slicing [42] extracts program statements relevant to a stack trace of a program crash. Barrier slicing [40]

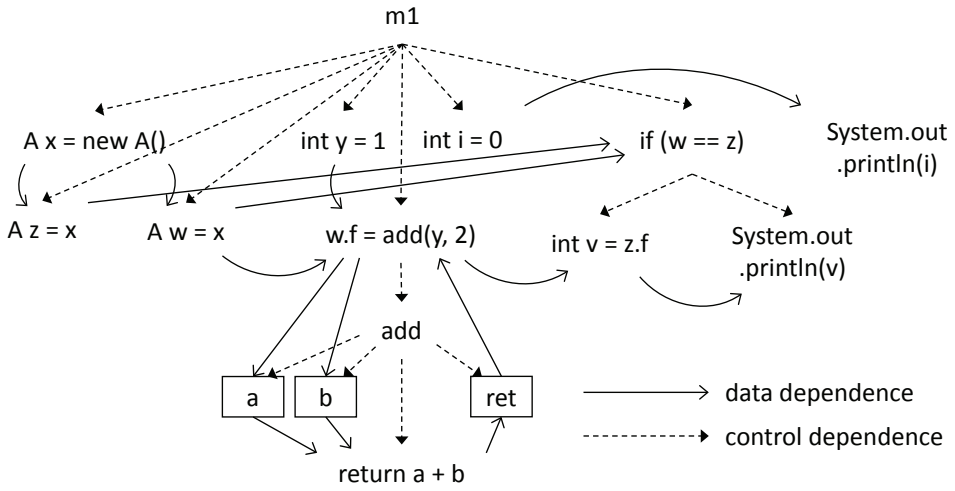

Figure 2.2: A system dependence graph for computing slices.

extracts a smaller slice by terminating graph traversal at barrier vertices selected by either developers or some conditions. Ceccato et al. [43] used barrier slicing to locate source code fragments implementing a certain process. Wang et al. [44] proposed a query language to search subgraphs that satisfy particular conditions in a SDG.

2.2.3 Thin slicing

Thin slicing [51] extracts all statements that produce data used at a slicing criterion. It reduces slice size and efforts for software maintenance. Sridharan et al. [51] proposed two computation methods of thin slicing: context-sensitive thin slicing and context-insensitive thin slicing. We address context-insensitive thin slicing because Sridharan et al. reported that it is difficult to apply context-sensitive thin slicing to large-scale systems due to its low-scalability.

A thin slice with respect to s is a set of statements that s are data-dependent on. Thin slicing defines three types of data dependencies as follows.

Intra-procedural data dependency

A statement s_2 is data dependent on s_1 if all of the following conditions hold.

1. s_1 defines a value of variable v.
2. s_2 uses the value of variable v.

3. There exists an execution path from s_1 to s_2 on which variable v is not redefined.

Thin slicing ignores data-flow into base pointers of field accesses. It focuses on the value flowing through the accesses. For example, with respect to `x = p.f`, thin slicing traces data dependencies on `f` and ignores data dependencies on `p`.

Inter-procedural data dependency

Thin slicing connects inter-procedural data dependency as follows:

- The vertices corresponding to formal parameters of callees are data dependent on the vertices corresponding to actual parameters of callers.
- The vertices that receive return values of callees are data dependent on the vertices corresponding to return values of callees.

Data dependency on heap

A statement which uses a filed (array) variable is data dependent on another statement which defines the filed (array) variable.

Computing thin slices

A thin slice is computed by graph traversal of a directed graph whose edges represent data dependencies. Figure 2.3 shows a dependence graph of Figure 2.1 program. The graph ignores control dependence edges and data dependence edges of base pointers (a edge from Lines 7 to 8 is ignored). A thin slice with respect to Line 11 is a set of Lines {5, 8, 10, 11, 16}. The thin slice reveals which statements produce a value used at Line 11.

Advantages of thin slicing

Thin slicing reveals which statements define a variable used at a slicing criterion because it focuses just on value flowing to reduce slice size. Sridharan et al. state that thin slicing is effective for many program understanding tasks because base pointer manipulation matters less than actual copying of the value through the heap.

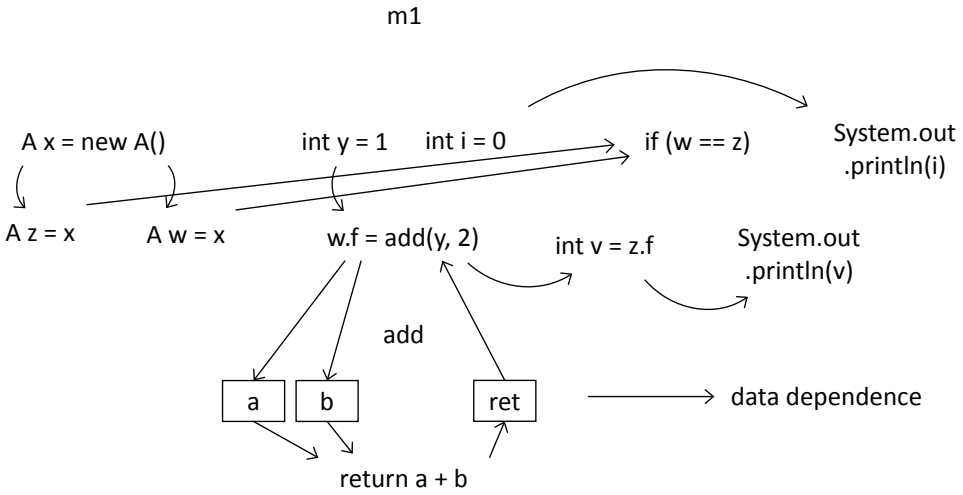

Figure 2.3: A dependence graph for computing thin slices.

2.2.4 Statistics on Slice Size

Existing studies have conducted empirical studies on statistics of slice size to evaluate how program slicing techniques are effective in general. Binkley et al. [56] investigated the average size of program slices. They reported that traditional program slicing extracts 30% of statements in C programs. Kashima [57] also reported average slice size in Java programs. He implemented a precise slicer for object-oriented systems [58] and showed that it extracts 9% of statements. Jász et al. [53] also conducted a similar experiment to evaluate their approach named Static Execute Before (SEB) relation, which is an approximation of traditional program slicing. They found that the average size of SEB is very close to that of traditional program slicing. Horwitz et al. [42] reported that a callstack-sensitive slice is about 0.31 time the size of the corresponding full slice on average. Our work follows these works to evaluate thin slicing because Sridharan et al. show only a few cases where thin slice sizes are measured.

2.3 Research Question

We conduct an empirical study on thin slice size to evaluate whether thin slicing is effective in general. We address two research questions as follows:

RQ1. Is the size of thin slice small enough on average?

RQ2. Are thin slices effective in program understanding?

Table 2.1: A list of bytecode instructions and their types.

Description	Type	Instructions
Read local variables	transfer	ILOAD, LLOAD, FLOAD, DLOAD, ALOAD
Write local variables	transfer	ISTORE, LSOTRE, FSTORE, DSTORE, ASTORE
Read field variables	transfer	GETFIELD, GETSTATIC
Write field variables	transfer	PUTFIELD, PUTSTATIC
Read array variables	transfer	IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD, SALOAD
Write array variables	transfer	IASTORE, LASTORE, FASTORE, DASTORE, AASTORE, BASTORE, CASTORE, SASTORE
Operand stack management	-	POP, POP2, DUP, DUP2, DUP_X1, DUP2_X1, DUP_X2, DUP2_X2, SWAP
Create objects	source	NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY
Computations	source and sink	IADD, LADD, FADD, DADD, ISUB, LSUB, FSUB, DSUB, IMUL, LMUL, FMUL, DMUL, IDIV, LDIV, FDIV, DDIV, IREM, LREM, FREM, DREM, INEG, LNEG, FNEG, DNEG, ISHL, LSHL, ISHR, LSHR, IUSHR, LUSHR, IAND, LAND, IOR, LOR, IXOR, LXOR, IINC, I2L, I2F, I2D, L2I, L2F, L2D, F2I, F2L, F2D, D2I, D2L, D2F, I2B, I2C, I2S, LCMP, FCMPL, FCMPG, DCMPL, DCMPG
Create constants	source	ICONST_M1, ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, LCONST_0, LCONST_1, FCONST_0, FCONST_1, FCONST_2, BIPUSH, SIPUSH, LDC, DCONST_0, DCONST_1, ACONST_NULL
Comparison	sink	IFEQ, IFNE, IFLT, IFGE, IFGT, IFLE, IFNULL, IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IF_ACMPEQ, IF_ACMPNE, IFNONNULL, TABLESWITCH, LOOKUPSWITCH
Method invocations	other	INVOKEVIRTUAL, INVOKEINTERFACE, INVOKESPECIAL, INVOKESTATIC
Method exits	transfer	IRETURN, LRETURN, FRETURN, DRETURN, ARETURN, RETURN

2.4 Implementation of Thin Slicing

We implemented a thin slicer for Java bytecode to answer the research questions. We analyzed not source code but rather bytecode because we can obtain information analyzed by a compiler without parsing source code. Table 2.1 shows a list of bytecode instructions. We construct a directed graph whose vertices represent bytecode instructions and edges represent data dependencies between instructions. Figure 2.4 shows an example of bytecode instructions that correspond to the program of Figure 2.1. We describe the detail of our data dependence analysis.

2.4.1 Intra-procedural data dependence analysis

Local variables and elements of an operand stack

Java Virtual Machine stores values and computation results into an operand stack. We analyze two types of data dependence edges: one is a edge between a local variable and an element of an operand stack, the other is a edge between elements of an operand stack.

Operand stack management instructions

Operand stack management instructions have no data dependence edge because our implementation computes data dependence edges considering the effect of those instructions. For example, when a return value of a method invocation is not used, a POP instruction disposes of the value. Our implementation assumes that there is no dependence edge from the method invocation.

2.4.2 Inter-procedural data dependence analysis

For each method call site that calls m, we find methods $m_1, m_2, \ldots, m_k$ that may be invoked by the call site. For each parameter of m_i, an actual in/out parameter vertex is created and connected to its corresponding formal parameter vertex. We use Variable-Type Analysis [59] to resolve dynamic binding.

2.4.3 Data dependence analysis on heap

A data dependence edge connects instructions that may read and write the same field or the same array. First, a GETSTATIC instruction which reads a field f is data dependent on PUTSTATIC instructions that write the same field f. Each `static` field is identified by its class name and field name. Second, a GETFIELD instruction is data dependent on PUTFIELD instructions when they may access the same field of the same object. We use object-sensitive pointer analysis [60] that is based on Andersen's pointer analysis [61]. Similarly, array load instructions are data dependent on array store instructions when they may access the same array.

2.4.4 Directed graph of bytecode instructions

Figure 2.5 shows a dependence graph for bytecode instructions of Figure 2.4. The vertices surrounded by a dotted line represent instructions in `add` method and the others represent instructions in `main` method.

2.5 Experiment

We compute thin slices and their metrics to answer our research questions.

2.5.1 Measuring metrics

We divide bytecode instructions into three types: *source* produces data, *sink* consumes data, and *transfer* propagate data without changing the value. Table shows our classification for all instructions. Some instructions are *source* and *sink*. For example, IADD, which adds two integers and stores the result, is *sink* that consumes two integers and it is also *sink* that produces the computation result for following instructions. The actual parameters of method calls are *sink* and return values are *source*.

We compute backward thin slices with respect to all sink instructions and forward thin slices with respect to all source instructions. This is because we expect that backward slices are effective for investigating how the values used by

```
Sample#m1#()V
  2: NEW
  3: DUP
  4: INVOKESPECIAL A#<init>()V
  5: ASTORE 0 (x)
  8: ALOAD 0 (x)
  9: ASTORE 1 (z)
 12: ICONST_1
 13: ISTORE 2 (y)
 16: ICONST_0
 17: ISTORE 3 (i)
 20: ALOAD 0 (x)
 21: ASTORE 4 (w)
 24: ALOAD 4 (w)
 25: ILOAD 2 (y)
 26: ICONST_2
 27: INVOKESTATIC Sample#add(II)I
 28: PUTFIELD A#f: int
 31: ALOAD 4 (w)
 32: ALOAD 1 (z)
 33: IF_ACMPNE L00044
 36: ALOAD 1 (z)
 37: GETFIELD A#f: int
 38: ISTORE 5 (v)
 41: GETSTATIC java/lang/System#out: java/io/PrintStream
 42: ILOAD 5 (v)
 43: INVOKEVIRTUAL java/io/PrintStream#println(I)V
 47: GETSTATIC java/lang/System#out: java/io/PrintStream
 48: ILOAD 3 (i)
 49: INVOKEVIRTUAL java/io/PrintStream#println(I)V
 52: RETURN
Sample#add#(II)I
  2: ILOAD 0 (a)
  3: ILOAD 1 (b)
  4: IADD
  5: IRETURN
```

Figure 2.4: Bytecode representation corresponding to the program of Figure 2.1.

method calls, conditional statements, and calculations are computed, and that forward slices are effective for investigating which statements use computation results and valuables. Transfers are excluded from slicing criteria because they just deliver data.

We define following notations for thin slices. The number of elements contained in S is denoted by $|S|$.

***Backward*(*v*)** A set of vertices computed by backward thin slicing with respect to a sink v.

***Source*(*v*)** A set of source vertices that are included in *Backward*(v).

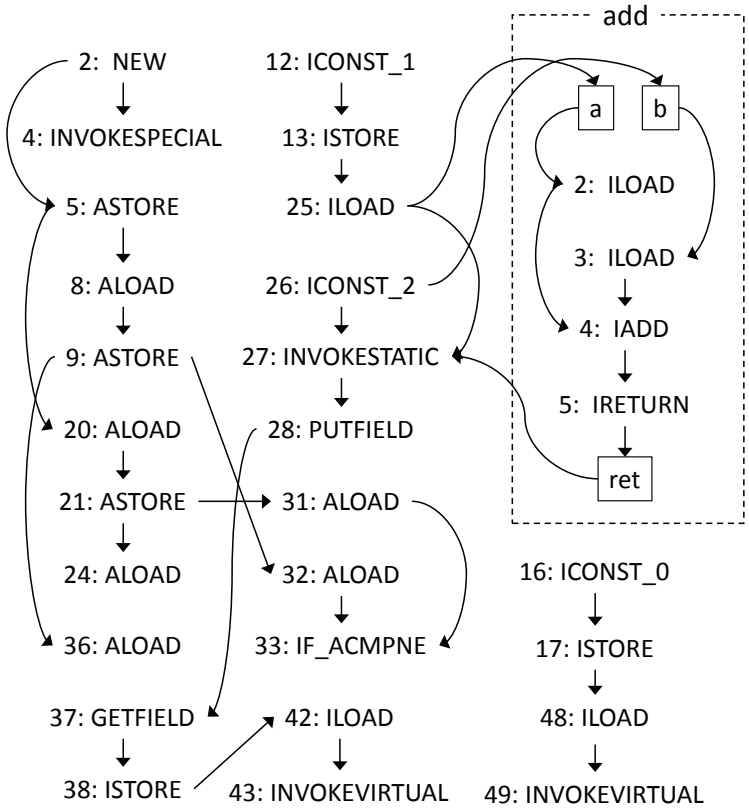

Figure 2.5: A dependence graph of bytecode instructions.

***Forward*(*w*)** A set of vertices computed by forward thin slicing with respect to a source w.

***Sink*(*w*)** A set of sink vertices that are included in $Forward(w)$.

***Method*(S_{ts})** A set of methods whose vertices are included in a thin slice S_{ts}.

***Class*(S_{ts})** A set of classes whose vertices are included in a thin slice S_{ts}.

***LOC*(S_{ts})** Source code lines contained within a thin slice S_{ts}.

We obtain line numbers from debug information embedded in class files. A pair of a method declaration vertex and its formal parameter vertex is counted as one source code line. Also, a pair of a method exit vertex and its return value vertex is counted as one source code line.

We demonstrate examples of measuring metrics. Let v be $43 : INVOKEVIRTUAL$.

$$\begin{aligned} Backward(v) = \{&12 : ICONST\_1, 13 : ISTORE, 25 : ILOAD, 26 : ICONST\_2, \\ &27 : INVOKESTATIC, 28 : PUTFIELD, 37 : GETFIELD, \\ &38 : ISTORE, 42 : ILOAD, 43 : INVOKEVIRTUAL, a, b, \\ &2 : ILOAD, 3 : ILOAD, 4 : IADD, 5 : IRETURN, ret\} \end{aligned}$$

$$Source(v) = \{12 : ICONST\_1, 13 : ISTORE, 4 : IADD\}$$

$$Method(Backward(v)) = \{main, add\}$$

$$Class(Backward(v)) = \{Sample\}$$

$$LOC(Backward(v)) = \{5, 8, 10, 11, 16\}$$

To answer RQ1, we compute $|Backward(v)|$ with respect to each sink v and $|Forward(w)|$ with respect to each source w. To answer RQ2, we compute following metrics for each sink v.

- $|Method(Backward(v))|$
- $|Class(Backward(v))|$
- $|Source(v)|$

Furthermore, we compute following metrics for each source w.

- $|Method(Forward(w))|$
- $|Class(Forward(w))|$
- $|Sink(w)|$

When $|Source(v)|$ is small, we can narrow down producers of data to a few by using thin slicing. Similarly, when $|Sink(w)|$ is small, we can narrow down consumers of data to a few. When $|Method(Backward(v))|$, $|Class(Backward(v))|$, $|Method(Forward(w))|$, $|Class(Forward(w))|$ are big, their thin slices lie on various methods or classes. In these cases, it is expected that thin slicing reduces the time required for identifying producers and consumers while considering method call relationships.

2.5.2 Subject programs

We analyzed seven Java programs in DaCapo benchmark 9.12 as shown in Table 2.2. They were compiled by JDK 1.6.0_45 on Windows 7 (64bit) with library classes. However, tomcat, pmd, xalan, and batik have no information about source code lines because their source code is unavailable.

Table 2.2: Subject programs.

Program	Classes	Methods	Vertices	Lines
tomcat	261	2,389	54,468	-
luindex	560	4,180	123,191	36,060
sunflow	657	4,609	190,526	43,974
avrora	1,838	9,304	211,343	68,936
pmd	2,369	16,439	448,722	-
xalan	2,805	22,377	815,861	-
batik	4,417	28,818	968,470	-

2.5.3 Results

RQ1

Table 2.3 shows the summary of measured metrics. The average size of backward and forward thin slices is about 2.1% on the seven programs. Binkley et al. [37] reported that the average size of traditional slices is about 30% in C programs. Also, Kashima [57] reported that the average size is 9% in Java programs. Some of his subject programs are same as our ones although his results do not include library classes. Considering these reports, we conclude that the average size of thin slices is small enough. For luindex, sunflow, and avrora, while the average of $|LOC(Backward(v))|$ was 0.3 point greater than that of $|Backward(v)|$, a correlation coefficient between them was 0.9996. From these results, we expect that source code lines of thin slices are also small.

However, thin slices are polarized into large and small ones. Figure 2.6 is cumulative frequency distribution whose horizontal axis is for the ratio of $|Backward(v)|$ to a program, and vertical axis is the number of thin slices. It shows that 60 to 80% of backward slices are less than 0.1% of a program while the rest are about maximum value of slices.

RQ2

Table 2.4 shows the ratio of thin slices that have a few producers and capture inter-procedural data-flow paths. These 10% slices can be effective for tracking data-flow because thin slicing tells producer values developers without traversing multiple methods.

Table 2.3: Summary of metrics

Subject		tomcat	luindex	sunflow	avrora	pmd	xalan	batik
Vertices		54,468	123,191	190,526	211,343	448,722	815,861	968,470
$\|Backward(v)\|$	max	922	12,820	34,512	30,458	40,784	62,957	93,810
	ratio(%)	1.7	10.4	18.1	14.4	9.1	7.7	9.7
	mean	55.2	3012.8	11264.9	6919.3	7108.6	12482.7	26159.3
	ratio(%)	0.10	2.4	5.9	3.3	1.6	1.5	2.7
$\|Forward(w)\|$	max	3,673	23,633	43,041	27,592	52,339	105,749	140,965
	ratio(%)	6.7	19.2	22.6	13.1	11.7	13.0	14.7
	mean	57.2	1434.0	7834.8	2386.0	6226.3	18481.9	28719.3
	ratio(%)	0.11	1.2	4.1	1.1	1.4	2.3	3.0
$\|LOC(Backward(v))\|$	max	-	4,678	7,144	10,602	-	-	-
	ratio(%)	13.0	16.2	15.4	-	-	-	
	mean	-	1102.6	1933.2	2425.6	-	-	-
	ratio(%)	3.1	4.4	3.5	-	-	-	
$\|Method(Backward(v))\|$	max	193	1,342	2,010	3,780	4,277	6,849	7,326
	mean	8.0	155.6	155.7	128.7	126.5	385.5	576.9
$\|Method(Forward(w))\|$	max	368	1,852	2,449	3,702	3,132	7,175	8,011
	mean	6.4	116.1	230.2	331.0	369.6	1267.1	1632.0
$\|Class(Backward(v))\|$	max	41	288	338	1,284	685	1,352	1,792
	mean	2.9	68.4	112.8	278.0	111.3	264.9	489.7
$\|Class(Forward(w))\|$	max	68	375	348	1,147	507	1,306	1,750
	mean	2.3	35.9	46.2	98.9	67.0	243.4	361.6
$\|Source(v)\|$	max	382	5,494	15,451	14,964	18,900	27,441	40,687
	mean	25.3	1371.4	5218.1	3470.6	3076.6	5215.6	11183.8
$\|Sink(w)\|$	max	1,610	9,216	17,369	11,689	19,425	41,423	52,334
	mean	25.4	537.6	3031.9	1021.4	2264.2	7238.5	10662.3

2.6 Conclusion of This Chapter

We measured thin slice size, the number of producer instructions, the number of methods included in the slice, and others. The results showed that the average size of thin slices is small enough. Furthermore, about 10% of thin slices can be effective for tracking data-flow because they have a few producers and capture inter-procedural paths. We expect that thin slicing can support developers to understand large-scale systems by visualizing inter-procedural data-flow paths. Ishio et al. [62] reported that visualization of data-flow paths reduced the time required for understanding.

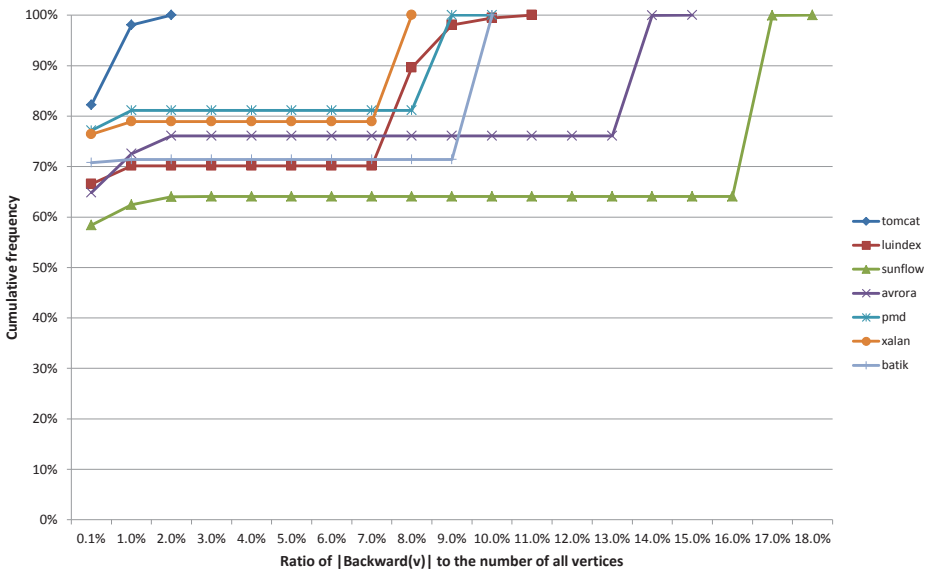

Figure 2.6: Cumulative frequency distribution for $|Backward(v)|$

Table 2.4: Ratio of thin slices whose $|Method(Backward(v))| \geq 2$ and $|Source(v)| \leq 3$.

	$\|Source(v)\|$			
Subject	1	2	3	total
tomcat	2.9	3.6	3.9	10.3
luindex	3.0	4.5	3.6	11.0
sunflow	1.8	2.5	2.4	6.7
avrora	3.5	3.7	2.5	9.7
pmd	2.8	3.5	4.1	10.5
xalan	2.7	3.4	3.3	9.4
batik	2.5	3.5	2.7	8.7

Chapter 3

Extracting Conditional Statements for Understanding Business Rules

3.1 Introduction

For the maintenance of a business system, developers must understand the business rules implemented in the system [48, 63, 64]. Business rules are classified into several types, such as computational business rules and constraints [63]. 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 [50]. 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 [32] is used to understand business rules [47–50, 64]. Cosentino et al. [50] 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 [50] 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.

This chapter tackles a problem that existing techniques include technical statements as pointed out by Cosentino [50]. 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 chapter, 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.

The contributions of this chapter 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.

3.2 Related Work

Sneed et al. [46] 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 [47], C/C++ [48], Java [49], respectively. Furthermore, Cosentino et al. [50] 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 [51] 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 [65] 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 [66] 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 understanding. Our technique focuses on partial control-flow paths for a given variable to understand computational business rules.

Dubinsky et al. [67] 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 [68] 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

Table 3.1: 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

execution. To overcome these challenges, their technique requires actual test cases and their execution results. Jaffar et al. [69] proposed a path-sensitive control-flow graph where a statement may be represented by multiple vertices. Thisgraph 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.

3.3 Motivating Example

3.3.1 Business rules implemented in source code

Throughout this chapter, 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 3.1(a) and 3.1(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 Figure 3.1. 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 3.1(a). Lines 7 through 9 examine a constraint between two input variables and cancel the computation if the

```
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 }
```

Figure 3.1: An example method implementing business rules

constraint is violated. The third step computes an output hour at lines 15 through 18, following the rules presented in Table 3.1(b).

Developers maintaining the system must recover Tables 3.1(a) and 3.1(b) from the source code in Figure 3.1 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*?*

3.3.2 Extraction of business rules by program slicing

Backward program slicing [32] 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 [70].

Figures 3.2(a) and 3.2(b) illustrate examples of a CFG and program depen-

dence 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 3.1(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 [65]). 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.

3.4 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

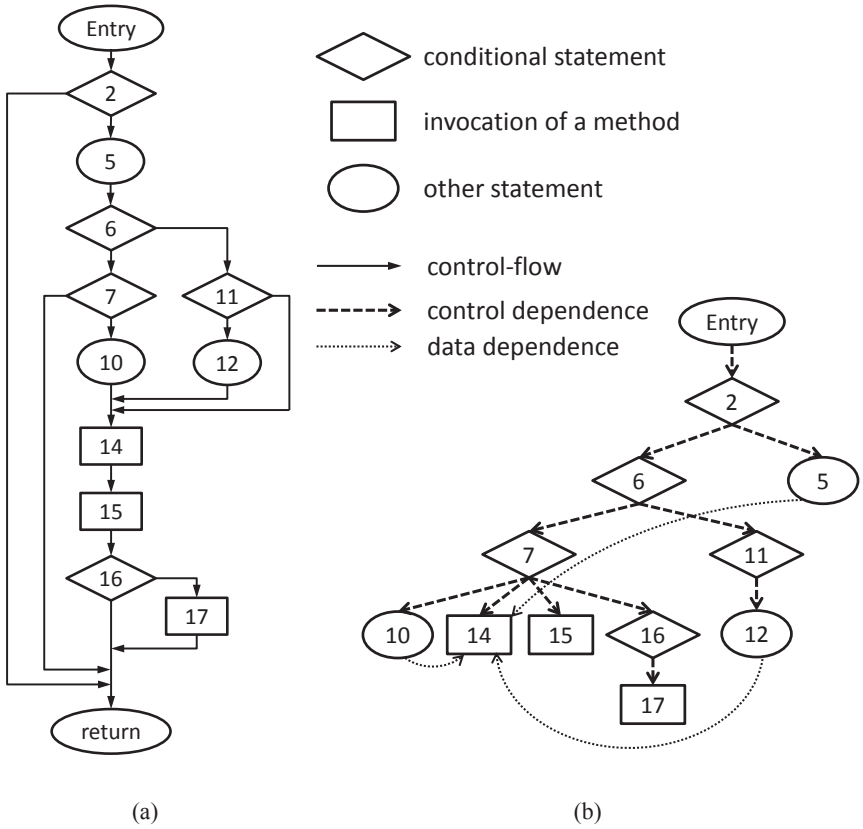

Figure 3.2: A control-flow graph (a) and program dependence graph (b) of Figure 3.1

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.

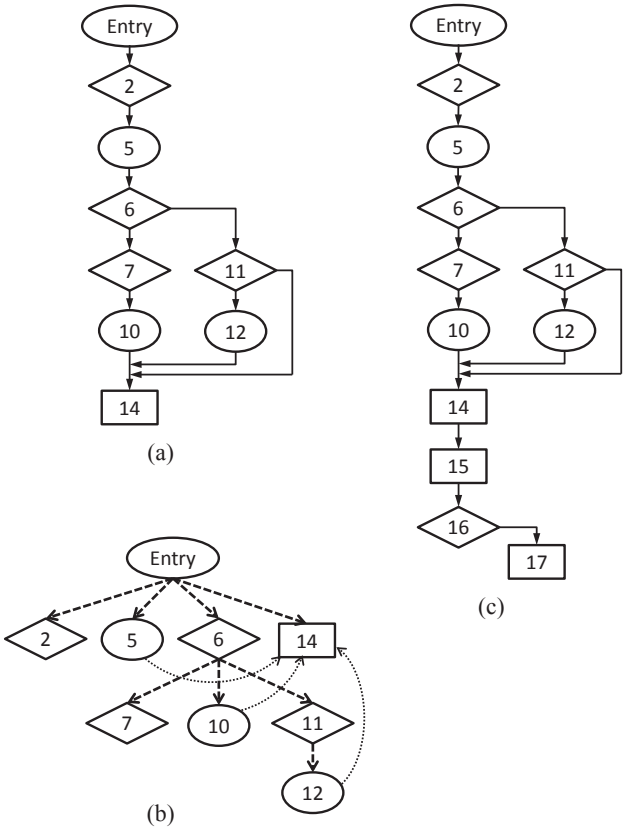

Figure 3.3: Three graphs to extract conditional statements: (a) is a subgraph of Figure 3.2(a) for setFee (line 14). (b) is a dependence graph extracted from (a). (c) is a subgraph of Figure 3.2(a) for setHour (lines 15 and 17).

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 [59] for our implementation.

3.4.1 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 repre-

sent control-flow paths [70]. 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 $x \xrightarrow{E*} x$.

Figure 3.3(a) is a sample subgraph of the CFG in Figure 3.2(a) with respect to setFee. For simplicity, the vertices in Figure 3.3 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 Section 3.3.

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. [71], we extract the following data-dependence edges.

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.

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.

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 Figure 3.1 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.

3.4.3 Extracting conditional statements

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

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

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 3.3(b) presents the dependence graph of the program in Figure 3.1 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 3.3(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 Figure 3.1. 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.

3.5 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?*

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.

3.6 Experiment with Human Subjects

3.6.1 Setup

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.

Tasks

The tasks used in our experiment were created from **MosP 4.0.0**[1], 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 3.2 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.

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

Table 3.2: 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

Table 3.3: 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

- 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. Table 3.3 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 Figure 3.1) 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.

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

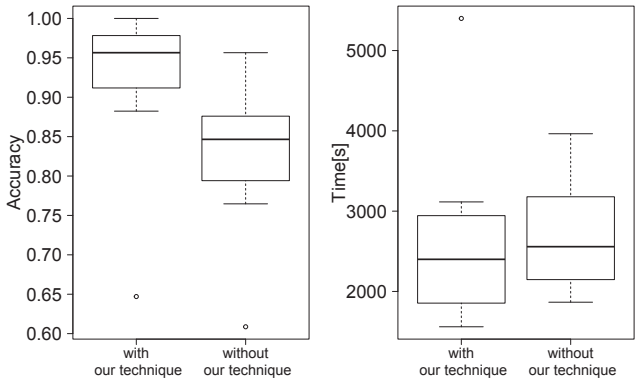

Figure 3.4: Comparison of the accuracy and time for tasks

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.

3.6.2 Results

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

The left box plot in Figure 3.4 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 [72], which measures the effect size for the test, indicated that the difference was large (the delta was 0.625) [73]. The improvement 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

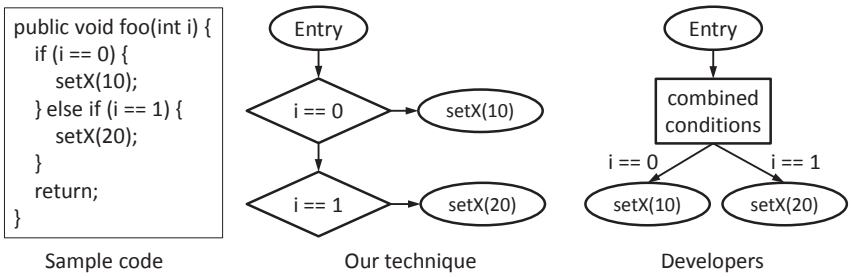

Figure 3.5: The difference between our technique and developers

to computational business rules.

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

The right box plot in Figure 3.4 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 and conceptual dependence. A simplified ex-

ample is illustrated in Figure 3.5. 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 Figure 3.5. 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.

3.7 Comparison with Program Slicing

3.7.1 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 3.4: 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

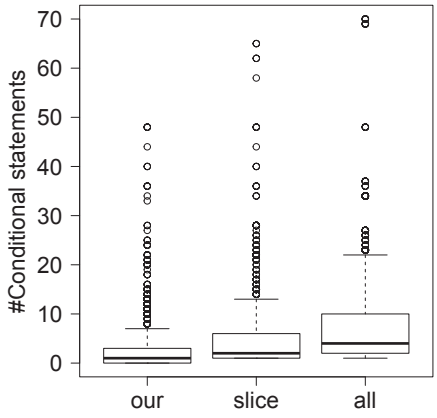

Figure 3.6: The number of conditional statements in MosP

3.7.2 Results

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

Table 3.4 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 3.4 represents the statistics of the number of conditional statements. Figures 3.6 and 3.7 plot the distributions of the number of conditional statements in MosP and the sales management system,

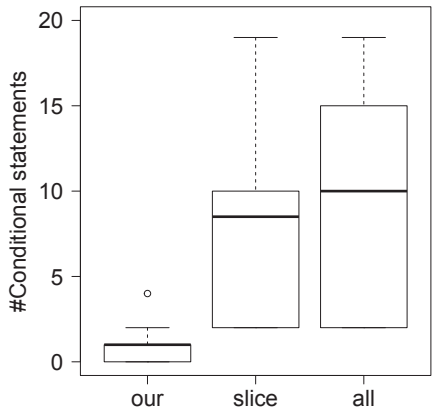

Figure 3.7: The number of conditional statements in the sales management system

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[2] 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 3.4, 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.

3.8 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

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

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 [3] 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 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.

3.9 Conclusion of This Chapter

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,

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

we plan to apply the proposed technique to other enterprise systems to evaluate the effectiveness of the proposed technique.

Chapter 4

Development of Program Analysis Tool for Java

4.1 Introduction

As shown in chapters 2 and 3, program analysis techniques play an important role which extracts useful information from software products and provides it for developers. For example, software metrics (such as lines of code and cyclomatic complexity) is used to ensure the software quality. Furthermore, control-flow, data dependence, control dependence, and method call relationships are also useful for understanding program structures.

One method to perform program analysis is parsing the source code to analyze syntax trees and symbol tables, like compilers. Existing analysis tools are designed to be able to analyze multiple programming languages [74]. Another method is analyzing compiled binaries. In Java, we can obtain package names of classes, types of method parameters, and others from bytecode though they are not directly appeared in the source code. Soot [75] adopts this method.

Although existing tools provide many features for program analysis, their users must understand the behavior and characteristics of algorithms. For example, because Soot provides rich options for analysis, users must have enough knowledge for deciding proper options. In the context of software engineering, it is difficult for users who do not know the detail of algorithms to utilize existing tools.

We developed SOBA (Simple Objects for Bytecode Analysis), which is a class library to provide basic features for bytecode analysis of Java. SOBA analyzes intra-procedural control-flow, data dependence, control dependence, method call relationships, and so on. It is easy to use without detailed knowledge of program analysis. The source code of SOBA is open as MIT license so that researchers can

use it for their program analysis studies. It is also used for the studies of chapters 2 and 3 in this dissertation.

4.2 Existing Tools and Our Motivation

Soot [75] and WALA [76] are bytecode analysis tools used for software engineering studies. These tools implement many algorithms that are useful for program analysis.

Although existing tools implement plenty algorithms, it is difficult to execute them and understand their execution results for those who do not understand the detail of algorithms. For example, when obtaining call graphs using Soot, we need to select a points-to analysis algorithm and implementation of binary decision tree for points-to analysis. Users are required to have detailed knowledge of algorithms because the accuracy and performance of analysis differ according to algorithms. Furthermore, existing tools have programming constraints due to their designs and implementation. For example, Soot's execution is divided in a set of different *packs* and each pack contains different *phases*. Users must understand their details and write programs following their constraints. Also, Soot is unsuitable for concurrent analysis because it is designed in the singleton pattern.

Figure 4.1 shows an example program which creates a call graph using Soot. This program is based on an example program shown in Soot's user guide [1]. We run the following command to execute this program. This command specifies target.jar as a target program, Class Hierarchy Analysis [77] as an algorithm for analyzing method call relationships, and classes referred by target.Main as analyzed classes.

```
java SootCallGraph -cp target.jar -whole program
                                    -p -cg.cha -app target.Main
```

Lines 3 through 15 in Figure 4.1 implements a pack and phase. This program adds a *SceneTransformer* object which implements a process for creating a call graph to "wjtp" pack which analyzes the whole program. Line 16 runs Soot's analysis. As a result, method call relationships are displayed at the console.

Figure 4.2 shows an example program which creates a call graph using WALA. This program is based on an example program shown in com.ibm.wala.examples.drivers.PDGCallGraph class. We run the following command to execute this program.

```
java WalaCallGraph target.jar
```

Lines 3 through 7 in Figure 4.2 specify analysis targets and options. Line 8 creates a object representing a call graph. Note that each method does not always correspond each vertex in this graph. Users must understand the detail of WALA's

[1] http://www.brics.dk/SootGuide/

```
 1: public class SootCallGraph {
 2:   public static void main(String[] args) {
 3:     PackManager.v().getPack("wjtp").add(new Transform("wjtp.myTrans",
                                                  new SceneTransformer() {
 4:       @Override
 5:       protected void internalTransform(String phaseName, Map options) {
 6:         CallGraph cg = Scene.v().getCallGraph();
 7:         for (Iterator<MethodOrMethodContext> callers = cg.sourceMethods();
                                                        callers.hasNext();) {
 8:           MethodOrMethodContext caller = callers.next();
 9:           for (Iterator<Edge> edges = cg.edgesOutOf(caller); edges.hasNext();) {
10:             Edge edge = edges.next();
12:             SootMethod srcMethod = edge.getSrc().method();
13:             SootMethod tgtMethod = edge.getTgt().method();
14:             System.out.println(srcMethod.toString() + " may call "
                                                     + tgtMethod.toString());
15:     }}}}));
16:     soot.Main.main(args);
17: }}
```

Figure 4.1: An example program which analyzes method call relationships using Soot .

design and select proper options. In WALA, call graphs are abstracted to compare analysis results and performance between different algorithms. However, users must understand the detail of algorithms to understand analysis results.

Soot and WALA are suitable for evaluating the performance between different algorithms by researchers who know the details of algorithms. However, in software engineering studies, the evaluation is not always required. The selection of algorithms is not important when researchers need basic information about target programs such as a list of classes, methods, and call graphs. In this situation, users need a simple tool which provides getter methods for obtaining analysis results when target programs are given. We designed and developed SOBA to meet this requirement.

4.3 SOBA

SOBA is a class library to implement program analysis for Java bytecode. Users can obtain following information about programs.

- A list of classes and methods in the target programs.
- Method call relationships.
- Intra-procedural control-flow, control dependence, and data dependencen.

```
 1: public class WalaCallGraph {
 2:   public static void main(String[] args) {
 3:     AnalysisScope scope = AnalysisScopeReader
                                  .makeJavaBinaryAnalysisScope(args[0], null);
 4:     ClassHierarchy cha = ClassHierarchy.make(scope);
 5:     Iterable<Entrypoint> entrypoints = Util.makeMainEntrypoints(scope, cha);
 6:     AnalysisOptions options = new AnalysisOptions(scope, entrypoints);
 7:     CallGraphBuilder builder = Util
                .makeZeroCFABuilder(options, new AnalysisCache(), cha, scope);
 8:     CallGraph cg = builder.makeCallGraph(options, null);
 9:     for (CGNode caller: cg) {
10:       for (Iterator<CGNode> callees = cg.getSuccNodes(caller); callees.hasNext();) {
11:         CGNode callee = callees.next();
12:         IMethod callerMethod = caller.getMethod();
13:         IMethod calleeMethod = callee.getMethod();
14:         System.out.println(callerMethod.toString() + " may call "
                                                       + calleeMethod.toString());
15: }}}}
```

Figure 4.2: An example program which analyzes method call relationships using WALA .

SOBA is designed to enable those who do not have the detailed knowledge of program analysis to easily obtain the above. The above information was often required in our research group.

SOBA provides only basic algorithms for Java analysis. Class Hierarchy Analysis [77], which analyzes method call relationships, resolves dynamic binding according to the language specification of Java. The algorithms for intra-procedural control-flow, control dependence, and data dependence are established as compiler optimization techniques. SOBA is a small-scale library compared with Soot and WALA because we consider that the implementation of these algorithms should be provided without the high learning cost. The algorithms for inter-procedural control-flow, dependence analysis, and points-to analysis are active research fields even now. SOBA does not provide the implementation of these techniques because users must understand the detail of algorithms and select proper options. We expect that users learn program analysis algorithms through the development using SOBA.

4.3.1 Characteristics of SOBA

When users specify jar/zip/class files or directories as analysis targets, SOBA analyzes them. SOBA can read zip files containing multiple programs because it recursively decompress them.

SOBA creates hierarchical data structures (expect for VTAResolver) shown

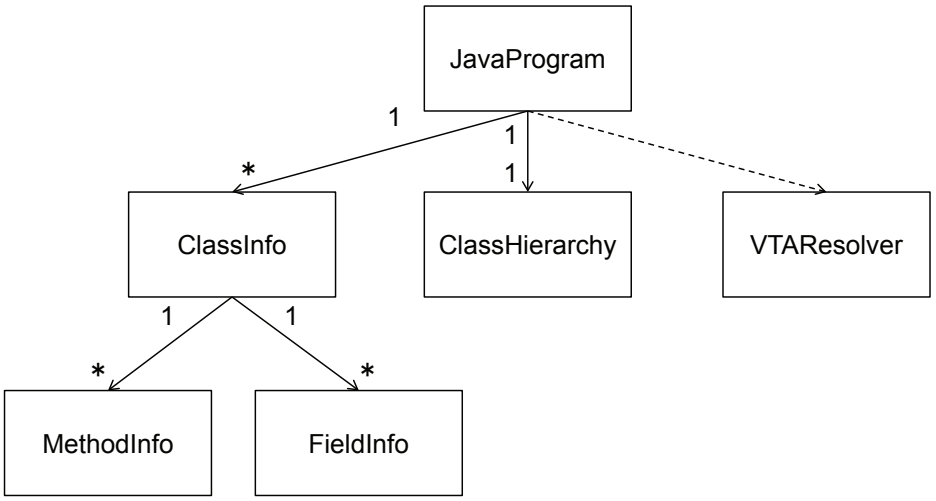

Figure 4.3: A class diagram of SOBA.

in Figure 4.3 Because VTAResolver, which is a class to analyze method call relationships, requires a lot of time and memory space, its instance is created by users' instruction. Users can access analysis results by invoking getter methods contained by each class. Table 4.1 shows the overview of main classes and their features of SOBA. When users invoke getter methods for control-flow, control dependence, and data dependence, their analysis results are cached in MethodInfo objects. Users can also access data structures of ASM (Tree API) because SOBA is a wrapper library of ASM. Tree API has detailed information such as string literals.

SOBA implements two algorithms for resolving dynamic binding: Class Hierarchy Analysis (CHA) [77] and Variable Type Analysis (VTA) [59]. CHA lists method candidates by analyzing the class hierarchy. VTA analyzes the class hierarchy and data types that may be assigned to receiver objects by tracking data-flow. While the original VTA computes reachable data types from the *main* method given by users, SOBA can analyze the whole program without giving the main method because it assumes that all data types are reachable for methods which are never invoked by others.

CHA is a basic algorithm which can exhaustively list method candidates for programs that use reflection because it computes method candidates according to the class hierarchy. However, even if CHA lists multiple candidates, they may be more limited in actual programs (e.g., invocations of List interface must be bound to ones of ArrayList). In these cases, users will require to exclude unlikely candi-

Table 4.1: Classes of SOBA.

Class name	Description	Feature
JavaProgram	Represents a whole program	Gets a list of classes (ClassInfo) declared in this program.
ClassInfo	Represents a class	Gets a class and package name. Gets a list of methods (MethodInfo) declared in this class. Gets a list of fields (FieldInfo) declared in this class.
MethodInfo	Represents a method	Gets a return type and signature of this method. Gets control-flow, control dependence, and data dependence. Invokes ASM Tree API.
FieldInfo	Represents a field	Gets a field name and type.
ClassHierarchy	Implementation of CHA [77]	Gets a superclass and subclasses of a specified class. Resolves dynamic binding.
VTAResolver	Implementation of VTA [59]	Resolves dynamic binding.

dates because CHA lists multiple candidates for all invocations. SOBA implements VTA, which takes linear time for program size, to apply large-scale programs.

SOBA analyzes data dependences between instructions that read/write local variables (except for array variables) and push/pop elements of an operand stack. SOBA does not analyze inter-procedural dependence because its analysis requires more advanced techniques such as points-to analysis.

SOBA labels classes to distinguish target classes and library classes when users specify them. We expect that users give labels to exclude information about library classes and call relationships within library classes from analysis results.

4.3.2 Example program

Analyzing method call relationships

Figure 4.4 shows an example program to analyze method call relationships. In this program, target programs are specified by arguments of a command. This program outputs strings that represent methods invoked by each method. Line 3 creates a JavaProgram object for target programs. Line 4 gets a ClassHierarchy object to resolve dynamic binding. for iterations in Lines 5 through 15 visit each class (ClassInfo) and method (MethodInfo). Line 7 lists instructions of method invocations and Line 8 resolve their callees using CHA. for iteration in Line 10

```
 1: public class SobaCallGraph {
 2:   public static void main(String[] args) {
 3:     JavaProgram program = new JavaProgram(ClasspathUtil.getClassList(args));
 4:     ClassHierarchy ch = program.getClassHierarchy();
 5:     for (ClassInfo c: program.getClasses()) {
 6:       for (MethodInfo m: c.getMethods()) {
 7:         for (CallSite cs: m.getCallSites()) {
 8:           MethodInfo[] callees = ch.resolveCall(cs);
 9:           if (callees.length > 0) {
10:             for (MethodInfo callee: callees) {
11:               System.out.println(" [inside] " + m.toLongString()
                                  + " may call " + callee.toLongString());
12:             }
13:           } else {
14:             System.out.println(" [outside] " + cs.toString());
15:  } } } } } }
```

Figure 4.4: An example program which analyzes method call relationships using SOBA.

outputs the callees. Line 14 outputs caller instructions because ClassHierarchy returns an empty array when callees are not included in target programs.

Figure 4.5 shows the execution result of the program described in Figure 4.4 for SOBA. Figure 4.5 lists signatures and return types of methods invoked by the main method. `<init>` represents an invocation of a constructor. The last line outputs `[outside]` because methods of Iterator class invoked in enhanced for-loops are not included in the analysis targets.

Analyzing data dependence

Figure 4.6 shows an example program which analyzes intra-procedural data dependence between instructions. Line 7 gets a DataDependence object. The iteration of Line 8 gets and outputs each data dependence edge which represents read/write relationships between instructions.

Figure 4.7 shows the execution result of the program described in Figure 4.6 for itself. A notation `i -> j` means that there exists a data dependence edge from the i-th instruction to the j-th instruction. Dependence edges on operand stacks and local variables are expressed as `STACK` and `LOCAL`, respectively. When the bytecode contains debug information, users can obtain variable names, types, and line numbers.

```
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/util/files/ClasspathUtil.getClassList(java/lang/String[]:files):
    soba/util/files/IClassList[]
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.<init>(soba/core/JavaProgram:this,
    soba/util/files/IClassList[]:lists): void
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.getClassHierarchy(soba/core/JavaProgram:this):
    soba/core/ClassHierarchy
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.getClasses(soba/core/JavaProgram:this):
    java/util/List
[outside] java/util/List.iterator()Ljava/util/Iterator; called by
  demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
                                    ...
```

Figure 4.5: Execution results of Figure 4.4.

```
 1:  public class DumpDataFlowEdge {
 2:    public static void main(String[] args) {
 3:      JavaProgram program = new JavaProgram(ClasspathUtil.getClassList(args));
 4:      for (ClassInfo c: program.getClasses()) {
 5:        for (MethodInfo m: c.getMethods()) {
 6:          System.out.println(m.toLongString());
 7:          DataDependence dd = m.getDataDependence();
 8:          for (DataFlowEdge e: dd.getEdges()) {
 9:            System.out.println(e.toString());
10:  } } } } }
```

Figure 4.6: An example program which analyzes data dependence using SOBA.

4.3.3 Example studies using SOBA

Kashima et al. [78] implemented inter-procedural dependence analysis using SOBA to visualize data and control dependence among methods. Kashima et al. [79] also implemented points-to analysis and program slicing. Although these studies are similar to a research filed of Soot and WALA, they implemented their analyzers based on SOBA to measure performance of them in detail.

Matsumura et al. [80] implemented a tool to replay states of local variables in Java programs using execution traces. The first author used SOBA to load subject programs, access data structures of ASM, and get intra-procedural control-flow graphs. Although this is the first time for the first author to analyze Java bytecode, he implemented the tool in short time without reading various papers about program analysis.

Kashiwabara [81] proposed a method to recommend verbs of methods using signatures, field names, and field types. This study obtained the above information

```
soba/example/dump/DumpDataFlowEdge
  .main(java/lang/String[]:args): void
PARAM -> 4 (LOCAL:0)
4 -> 5 (STACK:2)
2 -> 6 [1/2] (STACK:1)
5 -> 6 [2/2] (STACK:2)
2 -> 7 (STACK:0)
7 -> 10 (LOCAL:1)
  ...
```

Figure 4.7: Execution results of Figure 4.6.

Table 4.2: Comparison of the line number, class number, and command line options

Program	Comparison items	SOBA	Soot	WALA
Call relationships	# Lines	18	18	17
	# Imported classes	6	8	12
	Required command line options	Analysis targets	Analysis targets, Using CHA, Main class	Analysis targets
Data dependence and Control dependence	# Lines	17	15	27
	# Imported classes	7	9	21
	Required command line options	Analysis targets	Analysis targets	Analysis targets
# Total classes included in tools		116	3,248	1,575

from subject programs. Although the first author did not have knowledge about algorithms Soot implements and bytecode, she implemented analysis programs in short time using getter methods provided by SOBA.

4.4 Comparison with Soot and WALA

4.4.1 Programming

We compared the scale of programs using SOBA, Soot, and WALA. Table 4.2 shows the line number, class number, and command line options of two programs: one analyzes call relationships using CHA, and the other one analyzes data and control dependence. We open the source code of these programs that are also used to compare performance. We counted lines from the beginning to the end of a class excepting for blank lines. We used enhanced for-loops for classes that implement Iterable interface. "# Impoared classes" represents the number of required classes that are included in the package of each tool. "# Total classes included in tools" represents the number of classes that are not tests (their names do not end with "Test"). For WALA, we target only three projects: com.ibm.wala.core, com.ibm.wala.shrike, and com.ibm.wala.util.

Table 4.3: Comparison of features and their usage of each tool.

Feature	SOBA	Soot	WALA
Data structure	ASM Tree API	Jimple[2]	Statement object[3]
Loading analysis targets	Gives ClassPathUtil.getClassList jar/zip/class files	Specifies jar/class files using -cp option	Gives AnalysisScopeReader.makeJavaBinaryAnalysisScope jar/class files
Call relationships	Gets a ClassHierarchy object	Specifies main class and an algorithm using -whole-program -p cg.cha option, and invokes Scene.v().getCallGraph() method	Gives an AnalysisOptions object main class and gets an object implementing CallGraph interface
Data and control dependence	Gets a DataDependence object	Creates a HashMutablePDG object	Specifies options (DataDependenceOptions class , ControlDependenceOptions class) and creates a PDG object
Points-to analysis	None	Specifies -p cg.spark or -p cg.paddle options using command lines and invokes Scene.v().getPointsToAnalysis() method	Gets an object implementing PointerAnalysis interface
Inter-procedural dependence analysis	None	Creates a HashMutablePDG object or uses points-to analysis results	Creates a SDG object

While WALA's program which analyzes data and control dependence has larger lines, the other programs are almost same size. SOBA's programs require less classes than WALA's. Furthermore, total classes of SOBA are also fewer than the others because it provides only limited features. We conclude that SOBA reduces classes users must learn. Soot's program which analyzes call relationships requires command lines options to specify an algorithm.

4.4.2 Features

Table 4.3 lists features and their usage of each tool. All tools have corresponding features for specifying analysis targets, call relationships, and data/control dependence though their usage is different from each other. On the other hand, Soot and WALA have features for points-to analysis and inter-procedural dependence analysis while SOBA does not. However, users can implement context-insensitive inter-procedural analysis using SOBA.

While Soot and WALA require options to analyze call relationships and data/control dependence, SOBA does not. In call relationships analysis, Soot and WALA require main class given by users. When main class is given, tools analyze classes that are reachable from main class. This is not suitable for analyzing web applications in which specifying main class is difficult. Because SOBA analyzes all classes included in targets ignoring reachability, it may analyze needless classes for users. However, we consider that the negative impacts of analyzing needless classes is not significant in terms of the execution time and memory consumption because SOBA is implemented to be small and fast. While WALA provides advanced options to select algorithms of dependence analysis, SOBA provides only basic algorithms.

Table 4.4: Measured objects for performance comparison

Program	SOBA	Soot	WALA
Call relationships	Relationships between MethodInfo objects	Relationships between SootMethod objects	Relationships between objects implementing IMethod interface
Data and control dependence	Relationships between AbstractInsnNode objects of ASM Tree API	Relationships between PDGNode objects	Relationships between Statement objects

Table 4.5: Performance comparison of programs analyzing call relationships (time[ms] and memory[MB])

		SOBA		Soot		WALA	
Subjects	# Methods	Time	Memory	Time	Memory	Time	Memory
sunflow	4,828	1,894	285	10,744	1,915	2,250	151
avrora	10,073	2,085	344	12,005	1,068	2,965	540
pmd	19,523	2,630	455	20,868	654	4,405	362
h2	19,962	2,508	570	35,368	5,475	5,041	1,172
batik	36,063	3,522	801	35,369	8,006	8,041	637

4.4.3 Performance

We measured the execution time and memory consumption of two kinds of programs using SOBA, Soot, and WALA: one analyzes call relationships and the other analyzes data and control dependence. We selected five subject programs from DaCapo benchmark (version 9.12) [82]. Measurement targets are processes that obtain purpose information in the representation of each tool. Table 4.4 shows the detail of the representation.

Tables 4.5 and 4.6 show the performance of each tool. In this comparison, we use JDK1.8.0 as a standard library. We executed benchmark programs ten times on Intel Xeon E5-2690 2.90GHz to compare the average execution time and memory consumption. The result shows that SOBA is faster than Soot and WALA in call relationships analysis. While the memory consumption depends on subject programs in comparison of SOBA and WALA, SOBA requires less memory space than Soot. In data and control dependence analysis, SOBA requires less time and memory space than Soot and WALA. We conclude that SOBA is high performance because its features are limited.

Table 4.7 shows performance of call relationships analysis for large-scale programs. The VTA performance using Soot and WALA was not measured because Soot crashed in analysis and WALA does not provide VTA. SOBA requires less memory space than Soot and WALA because SOBA does not create a dependence graph to analyze call relationships while Soot and WALA have objects to represent all call relationships. Although VTA of SOBA requires a lot of memory space, it is able to analyze large-scale programs fast.

Table 4.6: Performance comparison of programs analyzing data and control dependence (time[ms] and memory[MB])

		SOBA		Soot		WALA	
Subjects	# Methods	Time	Memory	Time	Memory	Time	Memory
sunflow	4,828	1,894	285	10,744	1,915	2,250	151
avrora	10,073	2,085	344	12,005	1,068	2,965	540
pmd	19,523	2,630	455	20,868	654	4,405	362
h2	19,962	2,508	570	35,368	5,475	5,041	1,172
batik	36,063	3,522	801	35,369	8,006	8,041	637

Table 4.7: Performance for Eclipse 4.2 and JDK 1.7.0 (67,973 classes and 543,425 methods)

Algorithm	CHA			VTA
Tool	SOBA	Soot	WALA	SOBA
Execution time [ms]	15.5	65.8	27.3	95.5
Memory consumption [GB]	2.0	23.7	16.4	37.2

4.5 Conclusion of This Chapter

We developed SOBA, which is a class library to analyze Java bytecode. Its design enables those who do not have detailed knowledge of program analysis to easily obtain analysis results. We confirmed that SOBA is high performance in call relationships and data/control dependence analysis. Although our research group uses SOBA for our studies, we need feedback from other users to evolve it. Furthermore, we would like to evaluate whether SOBA has enough features for software engineering researchers.

Chapter 5

Conclusion

5.1 Summary of Studies

This dissertation described three studies on dependence analysis to tackle with the challenges of program understanding.

First, we conducted an empirical study to evaluate the effectiveness of thin slicing in terms of slice size. Our experiment showed that the average size of thin slices is about 2.1% on the seven programs. Furthermore, we found that 10% of thin slices can be effective for tracking inter-procedural data dependence. We believe that thin slicing can support developers to understand large-scale systems by visualizing data dependence.

Second, we developed a novel dependence analysis technique for understanding business rules. While existing techniques for business rules extraction include statements that do not correspond to the rules, our technique excludes those statements by constructing a partial control-flow graph. We evaluated whether this technique actually contributes to the performance of developers who extract business rules. A controlled experiment based on an actual process in one company showed that our technique enabled developers to more accurately identify conditional statements corresponding to rules without affecting the time required for the task.

Third, we developed SOBA, which is a program analysis tool for Java bytecode. Its design enables those who do not have detailed knowledge of program analysis to easily obtain basic information such as intra-procedural control-flow, data dependence, control dependence, and method call relationships. Researchers utilize SOBA for their studies on program analysis because its source code is open as MIT license.

5.2 Future Directions

Although existing studies and our first study on program slicing evaluated its general effectiveness by measuring slice size, we also need another evaluation which focuses on a specific understanding task. Researchers need to understand the characteristics of each technique and identify which technique is suitable for a certain task because program understanding includes various activities.

Some future work is needed for further support of understanding the source code which implements essentially complicated business rules. Even if we accurately describe the behavior of the source code, developers may not be able to understand the description. For example, although Chapter 3 introduced tables that describe business rules (Table 3.1), tables that describe essentially complicated rules would be much larger and more complicated. Future work includes a understandable expression way of business rules.

Bibliography

[1] Thomas A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, Vol. 28, No. 2, pp. 294–306, 1989.

[2] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 217–226, 1993.

[3] Janice Singer, Timothy C Lethbridge, Norman Vinson, Nicolas Anquetil, Norman G Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research*, pp. 21–36, 1997.

[4] Patrick Máder and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *Proceedings of the 28th International Conference on Software Maintenance*, pp. 171–180, 2012.

[5] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance*, Vol. 9, No. 5, pp. 299–327, 1997.

[6] Thomas D. LaToza, Gina Venolia, and R DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 492–501, 2006.

[7] Martha E. Crosby and Jan Stelovsky. How Do We Read Algorithms?: A Case Study. *IEEE Computer*, Vol. 23, No. 1, pp. 24–35, 1990.

[8] Bonita Sharif and Jonathan I. Maletic. An eye tracking study on camelcase and under-score identifier styles. In *Proceedings of the 18th International Conference on Program Comprehension*, pp. 196–205, 2010.

[9] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *Proceedings of the 17th International Conference on Program Comprehension*, pp. 158–167, 2009.

[10] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the 2012 Symposium on Eye Tracking Research and Applications*, pp. 381–384, 2012.

[11] Chris Parnin. Subvocalization - Toward hearing the inner thoughts of developers. In *Proceedings of the 19th International Conference on Program Comprehension*, pp. 197–200, 2011.

[12] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 378–389, 2014.

[13] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th International Conference on Automated Software Engineering*, pp. 43–52, 2010.

[14] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the 19th International Conference on Program Comprehension*, pp. 71–80, 2011.

[15] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 279–290, 2014.

[16] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st International Conference on Program Comprehension*, pp. 23–32, 2013.

[17] Bernhard Katzmarski and Rainer Koschke. Program complexity metrics and programmer opinions. In *Proceedings of the 20th International Conference on Program Comprehension*, pp. 17–26, 2012.

[18] Nadia Kasto and Jacqueline Whalley. Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the 15th Australasian Computing Education Conference*, pp. 59–65, 2013.

[19] Vallary Singh, Lori L. Pollock, Will Snipes, and Nicholas A Kraft. A Case Study of Program Comprehension Effort and Technical Debt Estimations. In *Proceedings of the 24th International Conference on Program Comprehension*, pp. 1–9, 2016.

[20] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 193–208, 2006.

[21] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn R. Chenm, and Gansner. Emden. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the 1999 International Conference on Software Maintenance*, pp. 50–59, 1999.

[22] Kenichi Kobayashi, Manabu Kamimura, Keisuke Yano, Koki Kato, and Akihiko Matsuo. SArF map: Visualizing software architecture from feature and layer viewpoints. In *Proceedings of the 21st International Conference on Program Comprehension*, pp. 43–52, 2013.

[23] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo. Feature-gathering dependency-based software clustering using Dedication and Modularity. In *Proceedings of the 28th International Conference on Software Maintenance*, pp. 462–471, 2012.

[24] Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 2–13, 1997.

[25] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. Using the Kleinberg Algorithm and Vector Space Model for Software System Clustering. In *Proceedings of the 18th International Conference on Program Comprehension*, pp. 180–189, 2010.

[26] Vassilios Tzerpos and RC Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pp. 258–267, 2000.

[27] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the 8th*

Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 326–337, 1993.

[28] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, Vol. 30, No. 5, pp. 63–70, 1997.

[29] Steven P. Reiss. Visualizing java in action. In *Proceedings of the 2003 Symposium on Software Visualization*, pp. 57–65, 2003.

[30] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3d. In *Proceedings of the 2006 Symposium on Software Visualization*, pp. 47–56, 2006.

[31] Fabian Beck, Oliver Moseler, Stephan Diehl, and Rey Günter D. In situ understanding of performance bottlenecks through visually augmented code. In *Proceedings of the 21st International Conference on Program Comprehension*, pp. 63–72, 2013.

[32] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352–357, 1984.

[33] Andrea De Lucia, Fasolino Anna Rita, and Munro Malcolm. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension*, pp. 9–18, 1996.

[34] Thomas D. LaToza and Brad a. Myers. Developers ask reachability questions. In *Proceedings of the 32nd International Conference on Software Engineering*, pp. 185–194, 2010.

[35] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2005.

[36] Paul Anderson, Thomas Reps, and Tim Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Transactions on Software Engineering*, Vol. 29, No. 8, pp. 721–733, 2003.

[37] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, Vol. 16, No. 2, pp. 1–32, 2007.

[38] Daniel Jackson and EJ Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd Symposium on Foundations of Software Engineering*, pp. 2–10, 1994.

[39] Thomas Reps and Genevieve Rosay. Precise Interprocedural Chopping. In *Proceedings of the 3rd Symposium on Foundations of Software Engineering*, pp. 41–52, 1995.

[40] Jens Krinke. Visualization of program dependence and slices. In *Proceedings of the 20th International Conference on Software Maintenance*, pp. 168–177, 2004.

[41] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proceedings of 8th International Workshop on Program Comprehension*, pp. 241–247, 2000.

[42] Susan Horwitz, Ben Liblit, and Marina Polishchuk. Better Debugging via Output Tracing and Callstack-Sensitive Slicing. *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, pp. 7–19, 2010.

[43] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. Barrier Slicing for Remote Software Trusting. In *Proceedings of the 7th International Working Conference on Source Code Analysis and Manipulation*, pp. 27–36, 2007.

[44] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the 25th International Conference on Automated Software Engineering*, p. 457, 2010.

[45] Hongyu Kuang, P Mader, and Hao Hu. Do data dependencies in source code complement call dependencies for understanding requirements traceability? In *Proceedings of the 28th International Conference on Software Maintenance*, pp. 181–190, 2012.

[46] Harry M. Sneed and Katalin Erdos. Extracting business rules from source code. In *Proceedings of the 4th International Workshop on Program Comprehension*, pp. 240–247, 1996.

[47] Hai Huang and WT Tsai. Business rule extraction from legacy code. In *Proceedings of the 20th Conference on Computer Software and Applications*, pp. 162–167, 1996.

[48] X Wang, J Sun, and X Yang. Business rules extraction from large legacy systems. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering*, pp. 249–253, 2004.

[49] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application. In *Proceedings of the 6th International Conference on Rules on the Web: Research and Applications*, pp. 17–31, 2012.

[50] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. Extracting business rules from COBOL: A model-based framework. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pp. 409–416, 2013.

[51] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th Conference on Programming Language Design and Implementation*, pp. 112–122, 2007.

[52] Xiaoran Wang, Lori Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. *Proceedings of the 18th Working Conference on Reverse Engineering*, pp. 35–44, 2011.

[53] Judit Jász, Árpad Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the 24th International Conference on Software Maintenance*, pp. 137–146, 2008.

[54] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices*, Vol. 23, No. 7, pp. 35–46, 1988.

[55] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engingeering*, Vol. 7, No. 1, pp. 49–76, 2002.

[56] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, Vol. 30, No. 1, pp. 3–es, 2007.

[57] Yu Kashima. *Study on Licensing and Program Understanding for Reuse Support*. PhD thesis, 2015.

[58] Christian Hammer and Gregor Snelting. An improved slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '04*, pp. 17–22, 2004.

[59] Vijay Sundaresan and Laurie Hendren. Practical virtual method call resolution for Java. In *Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 264–280, 2000.

[60] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 1, pp. 1–41, 2005.

[61] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[62] Takashi Ishio, Shogo Etsuda, and Katsuro Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading. In *Proceedings of the 20th International Conference on Program Comprehension*, pp. 37–46, 2012.

[63] Karl Wiegers and Joy Beatty. *Software Requirements*. Miscrosoft press, 2013.

[64] Harry M. Sneed. Extracting business logic from existing COBOL programs as a basis for redevelopment. In *Proceedings 9th International Workshop on Program Comprehension*, pp. 167–175, 2001.

[65] Keith Brian Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 751–761, 1991.

[66] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, Vol. 68, No. 1, pp. 45–64, 2003.

[67] Yael Dubinsky, Yishai Feldman, and Maayan Goldstein. Where is the business logic? In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 667–670, 2013.

[68] Josef Pichler. Specification extraction by symbolic execution. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pp. 462–466. Ieee, 2013.

[69] Joxan Jaffar and Vijayaraghavan Murali. A path-sensitively sliced control flow graph. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pp. 133–143, 2014.

[70] Frances E. Allen. Control flow analysis. *ACM Sigplan Notices*, Vol. 5, No. 7, pp. 1–19, 1970.

[71] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 345–387, 1989.

[72] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, Vol. 114, No. 3, pp. 494–509, 1993.

[73] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using T-Test and Cohen'Sd for Evaluating Group Differences on the NSSE and Other Surveys? In *Proceedings of the 2006 Annual Meeting of the Florida Association of Institutional Research*, pp. 1–33, 2006.

[74] Baroni Aline, Lucia and Abreu O Brito, E. An OCL-based formalization of the MOOSE metric suite. In *Proceedings of the 7th ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, 2003.

[75] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 13–23, 1999.

[76] WALA Project. WALA: T. J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[77] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pp. 77–101, 1995.

[78] Yu Kashima, Takashi Ishio, Shogo Etsuda, and Katsuro Inoue. Variable Data-Flow Graph for Lightweight Program Slicing and Visualization. *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 6, pp. 1194–1205, 2015.

[79] Yu Kashima, Takashi Ishio, and Katsuro Inoue. Comparison of Backward Slicing Techniques for Java. *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 1, pp. 119–130, 2015.

[80] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 253–257, 2014.

[81] Yuki Kashiwabara, Takashi Ishio, and Katsuro Inoue. Improvement in Method Verb Recommendation Technique using Association Rule Mining. *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 11, pp. 1982–1985, 2015.

[82] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-oriented Programming Systems, Languages, and Applications*, pp. 169–190, 2006.