# Study on Licensing and Program Understanding for Reuse Support

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2015

Yu Kashima

# Abstract

For developing reliable software, it is important to reuse existing software components. Reusable components are not only their developer's own, but also Open Source Software. In addition, there are hosting services to support developing and sharing open source software recently. As a result, it has been easier to search or distribute reusable components.

When reusing or distributing reusable components, there are a lot of concerns. In particular, *software license*, *retrieval for reusable components implementing a feature*, *extraction of a reusable component* are major important issues addressed in this dissertation.

Software license permits or forbids the usage of the software including reuse. If a developer violates a license statement of a reused component, the developer may have to stop developing or receive some legal actions. Therefore, the selection of the software license seems to affect to reuse activity, but not surveyed quantitatively. For encouraging active reuse activities, the impact of a software license should be known to the developers.

A developer needs to retrieve a reusable target using search engines, because the number of existing open source software is very large. Since major search engines often uses keyword-based search, a reuse target should include a keyword representing its feature. The targets of keyword-based search includes the source code of the software which contains names of identifiers. Identifier names are also important for understandability of the source code since a developer generally tries to guess a role of a program element from identifier names. However, unfortunately, not all developers are able to give appropriate names to identifiers, since a broader knowledge and a great deal of experience are necessary to define accurate names.

After a reusable target is found, reuse will complete if a feature of the target is incorporated into a user's product. However, it is not easy since a developer often has to extract a component implementing the reuse target feature from the found reusable target. For supporting the extraction, program slicing based techniques are proposed previously. In addition, there are several new program slicing techniques which are applicable for Java program language, e.g. *static execute before*, and *improved slicing*. However, these new techniques were not compared comprehensively, besides the comparative information is important for selecting a technique which is appropriate for the situation.

We believe that the targets of support for software reuse should include not only a user of a reuse component, but also a developer of a reusable component. Issues of licenses should be resolved by supporting a developer of reusable components, because software license is only decided by the developer. We resolved the license issue by reporting quantitative information which is useful for a license selection. Next, we resolved the retrieval issue from the viewpoint of identifier names since they have an important role in searching as described above. Identifier names in the component are naturally changed by only the developers. Therefore, this issue should be also resolved by supporting them. On the other hand, the extraction issue should be resolved by supporting a user of a reusable component, since reuse can be realized without burdening a component developer if the component user can apply an appropriate extraction technique.

First, we studied the impact of software licenses to reuse activity among open source software. In particular, we focused on copy-and-paste activity because copy-and-paste is a basic method for source code reuse. This study has shown frequent occurrence of copy-and-paste within the same license products. In addition, the study quantitatively has shown that permissive licensed files tend to be more copied than restrictive licensed ones. The result of this study gives a quantitative guide for the developer's license selection.

Secondly, we proposed an approach for building domain specific verb-object relationship dictionaries. The dictionary includes tuples consisting of (Verb, Direct Object, Indirect Object) which are extracted from identifiers in method signatures. In the experiment, we showed that the tuples in the dictionaries which are popular in the target domain or common Java programs. The result of the second study supports the appropriate naming and resolving the searching issue.

Thirdly, we performed a comparative study of four backward program slicing techniques for Java. The techniques include *static execute before*, *context-insensitive slicing*, *hybrid technique of static execute before and context-insensitive slicing*, and *improved slicing*. The comparison shows that the hybrid technique has the scalability for analyzing a large system, and outputs a slice whose size is 25 percent smaller than static execute before, which is the most lightweight technique, on average. On the other hand, improved slicing has the scalability for analyzing middle size applications, and is 31 percent smaller than compared to the hybrid technique. The results of the third study help selection of the program slicing techniques and extraction of a reusable component.

We believe that the results of these studies support developers to determinate a license of reusable components, to search a product implementing a target feature, and to extract a reusable component from the product.

# List of Major Publications

## Academic Journal

1. Yu Kashima, Yasuhiro Hayase, Yuki Manabe, Katsuro Inoue. Building Domain Specific Dictionaries of Verb-Object Relation from Source Code (in Japanese). IPSJ Journal, Vol.54, No.2, pp.857–869. 2013.

2. Yu Kashima, Takashi Ishio, Katsuro Inoue. Comparison of Backward Slicing Techniques for Java. IEICE Transactions, Vol. E98-D, No. 1, January, 2015.

## International Conference

1. Yu Kashima, Yasuhiro Hayase, Norihiro Yoshida, Yuki Manabe, Katsuro Inoue. An investigation into the impact of software licenses on copy-and-paste reuse among OSS projects. In Proceedings of the 18th IEEE Working Conference on Reverse Engineering, pp.28-32. Limerick, Ireland, October, 2011.

2. Yu Kashima, Yasuhiro Hayase, Norihiro Yoshida, Yuki Manabe and Katsuro Inoue. A preliminary Study on Impact of Software Licenses on Copy-and-Paste Reuse. In Proceedings of the 2nd International Workshop on Empirical Software Engineering in Practice, pp.47-52. Nara, Japan, December, 2010.

3. Yasuhiro Hayase, Yu Kashima, Yuki Manabe and Katsuro Inoue. Building Domain Specific Dictionary of Verb-Object Relation from Source Code. In Proceedings of the 15th IEEE European Conference on Software Maintenance and Reengineering, pp.93-100. Oldenburg, Germany, March, 2011.

# Acknowledgement

I am most indebted to my supervisor Professor Katsuro Inoue for his continuous support and supervision over the years. Without his help, experience and advice, this thesis would never have reached completion.

I am also very grateful to Assistant Professor Takashi Ishio for a lot of valuable support, supervisions, and helpful criticism of this thesis.

I want to thank Assistant Professor Yasuhiro Hayase in University of Tsukuba, Associate Professor Norihiro Yoshida in Nagoya University, and Assistant Professor Yuki Manabe in Kumamoto University. Their comments, criticism and advice have helped guide and shape the development of this thesis.

I would like to express my gratitude to Associate Professor Makoto Matsushita for his worth supports.

I would also like to express my gratitude to Professor Shinji Kusumoto and Professor Yasushi Yagi for his valuable comments on this thesis.

I would like to express my gratitude to all members of the Department of Computer Science for their guidance, especially Professor Toshimitsu Masuzawa.

Thanks are also due to many friends in the Department of Computer Science, especially students in Inoue Laboratory.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Software Reuse

Software reuse makes valuable contributions in developing reliable software rapidly. In order to improve the quality of software in the world, it is desirable that software is actively distributed via reuse.

There are various forms of reuse, including the following examples:

**Reuse as an External Tool** uses existing software as an external tool, e.g. combining *sort* command through a unix shell pipe. The advantages of this method are easiness of avoiding a license issue described later, and treating the reuse target as a black box. The disadvantage is the lack of flexibility.

**Reuse as a Library** links to the existing software dynamically or statically, and then use a feature of the linked software from their accessible functions such as API. For example, we can use various container classes for Java included in *Apache Commons Collections* [1]. The advantage of this method is convenient since the library functions can be accessed by the software. The disadvantage is that this method requires a large effort of developers for making their software reusable as a library.

**Source Code Reuse** embeds source code of a reuse target in the developing software, e.g. copy-and-paste the reuse target source code. The advantage of this method is flexibility. Developers can modify the reused source code as they want. The disadvantage of this method is that the availability of source code is dependent on the license of the reuse target.

In this dissertation, we would like to support source code reuse since this method is the most flexible. In addition, recently, places for sharing open source software (OSS) are increased, e.g. hosting services including SourceForge.net [2], GitHub [3], and etc. As a result, it has been easier to search or distribute components which can be used for source code reuse.

However, there is much of the issues about source code reuse. The followings are the major issues addressed in this dissertation.

**Software License** Software reuse is permitted or forbidden by the software license. For example, a typical software license of commercial software forbids rent, copy, redistribute, and reverse engineering of the software. On the contrary, software license for open source software generally permits those activities. The Open Source Initiative [4] publishes open source definition, which defines open source software as the ones distributed under a license which satisfies specific conditions [5]. The conditions include availability of the source code, allowing of modifications, derived works, and redistribution of the software.

There are various licenses proposed for OSS [4]. In particular, conditions of source code reuse are different for each license. The followings are examples:

**BSD 3-Clause License [6]** Software distributed under this license can be reused if a derivative work retains copyright notice, disclaimer of warranty, and the full text of the license

**Apache License [7]** Software distributed under this license can be reused if a derivative work retains notices of copyright, disclaimer of warranty, patents, trademarks, and modification from the original work.

**GNU General Public License [8]** If software under this license are distributed as a binary or a source code, source files of the software must be available to get. A derivative work using software under this license, which includes dynamic link, static link, and source code reuse must be distributed under the same license, i.e. GNU General Public License. Because of this feature, GNU General Public License is so called a *copyleft* license.

If a developer violates the condition of a license, the developer might have to stop the development or receive some legal actions. In the past, Epson violated the GNU General Public License of the *gettext* library used in a product. Because of this violation, Epson made the source code of the product public [9]. Similarly, the game *ICO* caused a license violation because this game used *libarc* library which was distributed under the GNU General Public License [10].

Qualitatively, selection of the software license seems to affect to reuse activity. Therefore, for encouraging active reuse activities, the impact of a software license should be known to the developers. However, no quantitative investigation was performed in the past.

**Retrieval for Reusable Components implementing a Feature** Nowadays, the number of open source software is very large, e.g. the number of repositories in GitHub has been overred 10 million in 2013 [11]. Therefore, developers usually use a search engine to find a reusable target. Although code search techniques, which identify similar source code to a given source code fragment, were proposed in academia [12, 13], major search engines which include web search engines and hosting services' ones currently use keyword-based search, Therefore, it is desirable that a reuse target includes a keyword representing its feature.

The targets of keyword-based search are not only the software names, the descriptions of the software, but also the source code of the software which includes comments and names of identifiers. In particular, a developer generally tries to guess a role of a program element from identifier names [14, 15]. Therefore, appropriate naming should improve not only the precision of keyword based search, but also the understandability of the source code. However, unfortunately, not all developers are able to give appropriate names to identifiers, since a broader knowledge and a great deal of experience are necessary to define accurate names.

**Extraction of a Reusable Component** After a reusable target is found, reuse will complete if a feature of the target is incorporated into a user's product. This is easy if the reuse target implements the feature in a modular way. However, naturally, not all of the OSS are modular about reusable features. In addition, the cost of making software modular seems very large [16].

Another approach is that developing a new reusable component from existing software. This process consists of locating a reuse target feature, extracting components implementing the feature, and restructuring them to a reusable component. Although developers have performed this process manually [17], it should be a tool supported to boost reuse activity.

*Program Slicing* [18], which is an analysis technique that developers use to extract statements related to a specific behavior of interest, has a great potential for supporting this approach. Lanubile et al. [19] proposed making a new reusable program from existing software based on program slicing [18]. Similarly, Komondoor [20] and Marx et al.[21] proposed approaches for component extraction based on program slicing technique, respectively.

Recently, there are several new program slicing techniques which are applicable to Java programming language, e.g. *static execute before* [22, 23], and *improved slicing* [24]. However, these new techniques were not compared comprehensively, besides the comparative information is important for selecting a technique which is appropriate for the situation.

## 1.2 Contributions to Reuse Support

We believe that support for software reuse should target at not only a user of reuse component, but also a developer of reusable components. Issues of licenses should be resolved by supporting a developer of reusable components, because software license is only decided by the developer. We resolved the license issue by reporting quantitative information which is useful for a license selection. Next, we resolved the searching issue from the viewpoint of identifier names since they have an important role in searching as described above. Identifier names in the component are naturally changed by only the developers. Therefore, this issue should be also resolved by supporting them. On the other hand, the last issue should be resolved by supporting a user of reusable component, since reuse can be realized without burdening a component developer if the component user can apply an appropriate extraction technique.

The following subsections describe our contributions.

### 1.2.1 Contributions to Software License Issue

For resolving the first issue, we investigated the relationship of software license and copy-and-paste activity among OSS projects. The investigation targets of this study were packages including Debian/GNU Linux 5.0.2 [25] and software hosting in SourceForge.net [2]. Software licenses of the source file were identified by Ninka [26] which identifies a license from a license statement in a source file. Copy-and-Paste activity was estimated by code clone analysis with CCFinderX [27] which detects source code fragments similar to other code fragments.

This study concludes the following two things quantitatively. First, copy-and-paste activity often occurs in products using the same license, Second, a product using the GNU Public License is significantly less copy-and-pasted to a product using other license product. These were known as qualitative, but not known as quantitative. We believe that the result of this study helps license selection by developers.

### 1.2.2 Contributions to Retrieval Issue

As described above, we believe accurate identifier naming helps to resolve the second issue. In previous research, our research group built a dictionary which includes the abstract-concrete relationship of a noun used in identifier names [28]. The dictionary seems to help naming of class names and variable names which consist of nouns. However, it seems not helpful for naming a method because a method name includes a noun, a verb, and a combination of verb and noun.

For showing useful information about a method name, we focused on verb-object relationship. We proposed an approach for building dictionary about verb-object relationships for a specific domain. First, source files are collected from software products which target a certain domain. Verb-object relationships were collected from method signatures in the set of existing software by a pattern matching approach using 29 extraction patterns that we have manually created. The collected verb-object relationships are filtered by the number of software, including the relationship in order to remove relations which are specific the software but not the domain. As a result, the domain specific verb-object relationship dictionary is made.

We evaluated the dictionary via an interpersonal experiment. In the experiment, six subjects evaluated the verb-object relations in four domain dictionaries which were made from 38 software. The result of the experiment shows that the relationships are popular on the target domain or common Java program, which indicate the effectiveness of our approach. We believe that the result of this study helps appropriate naming and resolving searching issue.

### 1.2.3 Contributions to Extraction Issue

As described above, program slicing is effective for extracting a reusable component. To be technically in detail, program slicing extracts a set of statements that may affect the value of a variable in a developer-specified statement. This technique has many applications other than reuse, including program comprehension [29], debugging [30, 31], and maintenance [32], etc.

Currently, system dependence graph (SDG) -based program slicing technique [33] is widely used. SDG is a directed graph which represents control and data dependencies between program statements. There are many kinds of SDG: e.g. SDG for representing Java program [34] and data-flow via the fields of objects [35, 36, 24], and representing exception handling in SDG [37]. On the other hand, as a program slicing technique but not using SDG, studies [22, 23] proposed static execute before/after analysis. This technique is less accurate than an SDG-based technique, but lightweight and scalable.

Tailoring program slicing for developers' and researchers' needs is an important issue. Java is the most popular programming language, in both open source [38] and industrial software development [39]; however, accuracy and scalability of Java program slicing techniques have not yet been investigated. Binkley et al. [40] evaluate program slicing for C/C++. They compare slices obtained with various configurations of CodeSurfer [41]. Jasz et al. [22] compare static execute before analysis and program slicing for C/C++. Beszedes et al. [23] compare static execute after analysis with forward program slicing in C/C++ and Java. They did not include a simple backward program slicing technique and static execute before analysis for Java, in the comparison. Moreover, improved slicing [24], an advanced slicing technique for Java, has not been evaluated with practical applications.

In this study, we compared the following four program slicing techniques:

- Static execute before [22]
- Context-insensitive slicing [40]
- Hybrid technique of static execute before and context-insensitive slicing
- Improved slicing [24]

In the experiment, we executed these slicing techniques to eight programs in Dacapo Benchmarks with two different configurations. In the first configuration, slicing targets were only application source code without library source code. The first configuration supposes a situation that a developer wants to save the cost of analyzing libraries, or a situation that libraries are unavailable. In the second configuration, slicing targets were the whole system including libraries. The second configuration supposes a situation that a developer wants to get a precise result. Furthermore, we performed an additional experiment that evaluating the scalability of the improved slicing since this technique could not analyze the target in the second configuration, although the technique showed the best precision in the first configuration.

The results of the experiments show that the hybrid technique had good scalability, which was achieved with a small cost increase over context insensitive slicing. A hybrid slice is 25 percent smaller than the slice by static execute before. Moreover, a hybrid slice is sometimes significantly smaller than a context insensitive slice, because hybrid technique considers feasible control-flow paths from static execute before. When developers need to analyze a large program or a program including a library, our results indicate that hybrid technique is suitable.

On the other hand, our results show that improved slicing is the most accurate of the four slicing techniques. A slice by improved slicing contains 22 percent of static execute before and 69 percent of hybrid technique. However, improved slicer does not have the scalability required to analyze a large program, such as a whole system including a Java development kit library. When developers need to analyze a middle-size program or a subsystem, our results indicate that improved slicing is suitable.

We believe that these results help the selection of the program slicing which is appropriate to a situation for a developer who wants to extract a component from an existing system.

## 1.3 Overview of the Dissertation

The rest of the dissertation is organized as follows: Chapter 2 reports an investigation of the impact of software license on copy-and-paste. Chapter3 describes an approach for building dictionary including verb-object relationships. Chapter 4 reports a comparison of backward slicing techniques for Java. Finally, Chapter 5 concludes our study and shows the future work.

# Chapter 2

# An Investigation into the Impact of Software Licenses on Copy-and-Paste Reuse among OSS Projects

## 2.1 Introduction

The source code of OSS is available to anyone to modify or redistribute. Considering the growth in OSS development [42], software developers today have a huge amount of OSS source code available for reuse.

Anyone who obtains or uses a software product must adhere to its license, which expresses the intent of the copyright holder. Several OSS licenses require derivative works to apply the same license of the original product, i.e. copyleft [43]. Since different right holders have different intents, many different OSS licenses are used. The license of an OSS product is usually determined at the outset of product development, and is rarely changed during the development process. To change the license, all copyright holders of the product must agree to the change.

Software reuse is recognized as a way of reducing development cost and improving product quality. Reuse can be defined at several levels of granularity; from simple copy-and-paste (CnP) of code snippets, to the inclusion of entire blocks of code, and even to the inclusion of a whole product. A developer who reuses a software product must adhere to the license of the reused product, since software reuse is just another form of use. Additionally, developers must take care not to violate the license of the product under development when reusing other software.

Since different OSS licenses have different restrictions on reuse [44], the software license may affect the frequency of code reuse. For instance, source code distributed under a copyleft license may be reused less frequently and in a smaller range of software products, compared with source code under a permissive license.

Therefore, previously we performed a preliminary study on CnP reuse using a small Java source file set from the viewpoint of software licenses [45]. The results of the investigation indicate that the license of a source file affects the frequency of reuse.

This study presents the results of a large scale quantitative study on the relation between software licenses and CnP reuses based on the following two research questions.

- **RQ1** Is source code distributed under a permissive license reused more frequently than that distributed under a restrictive license?

- **RQ2** Which type of licensed source code more frequently is imported to source code of other OSS products?

We performed two experiments for the RQs. The first experiment corresponds to both of RQ1 and RQ2, and the second experiment corresponds to RQ1. As the target of the experiments, we used source files of actual software products. One data set contains C/C++ source files from the main section of Debian GNU/Linux lenny [25], while the other set contains C/C++ source files randomly sampled from SourceForge.net [2]. To detect copy-and-pasted code fragments, CCFinderX [27] is used.

The results of the experiment show that source code distributed under the permissive licenses, is more frequently reused than restrictive license code. On the other hand, irrespective of the license type, source code is frequently reused in source code under GNU General Public License version 2 or later. The second experiment examines the statistical impact of the license of a reused file for the frequency of CnP. The results show that the licenses significantly affect the frequency.

The rest of this paper is organized as follows. Section 2.3 discusses the design and results of the two experiments, with an interpretation thereof. Section 2.4 discusses the validity of the experiment. Finally, Section 2.5 summarizes our study.

## 2.2 Background

### 2.2.1 OSS License

Nowadays, many open-source licenses are used; for example, the Open Source Initiative has officially approved 70 licenses [46]. This section discusses four of the most commonly used licenses, *BSD 3-clause license (BSD3)*, *MIT License (MIT)*, *Apache License 2.0 (Apachev2)* and *GNU General Public License version 2 (GPLv2)* from the perspective of a developer creating a derivative product. From the point of view of a developer, different licenses signify different restrictions as follows:

**BSD3** When a developer creates a derivative work from a *BSD3* product, the developer should retain the copyright notice and the full text of the license.

**MIT** Similar to the case of *BSD3*, when a developer creates a derivative work from a *MIT* product, the developer should retain the copyright notice and its permission notice.

**Apachev2** If a developer creates a derivative work from an *Apachev2* product, all copyrights, patents, trademarks, and attribution notices should be retained in the new product. Moreover, changed files should also have notices of the modifications.

**GPLv2** When a developer creates a derivative from a *GPLv2* product, the whole derivation must be distributed under the *GPLv2* and, changed files should have notices of any modifications.

### 2.2.2 OSS License for Copy-and-Paste Reuse

Software reuse is recognized as a practical method for developing high quality software with low cost. A developer can reuse the source code of OSS products since such source code is easily available.

Copy-and-paste (CnP) is one of the most frequently performed methods of reuse. In CnP reuse, source code is copied, modified if needed, and finally, used as part of the new product [44, 47].

When reusing existing software, the restrictions of both the license of the product being reused and that of the developing product must be adhered to. In case the two licenses are incompatible, both cannot be satisfied simultaneously. For example, Apache license version 2 (*Apachev2*) products

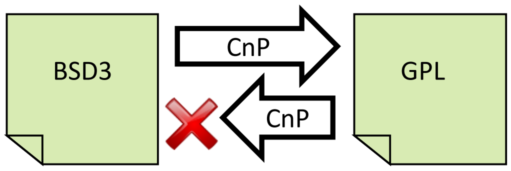


Figure 2.1: Reusing Source Code in a Different Licensed Product

cannot be incorporated into GPL version 2 *(GPLv2)* products, since several requirements in *Apachev2* conflict with a clause in *GPLv2* [48].

Furthermore, even if the licenses do not conflict, an OSS product cannot be reused if the license of the product being developed cannot be changed. For example, *GPLv2* source code cannot be incorporated into a *BSD3* product. In contrast, *BSD3* source code can be incorporated into a *GPLv2* product because the restrictions of *BSD3* are included in those of *GPLv2* (Figure 2.1).

For these reasons, the license of the reused product is a big concern in software reuse. The reusability of software depends not only on functionality and quality, but also on the license of the product; i.e. source code distributed under a permissive license is expected to be reused more frequently than that under a restrictive license, qualitatively.

## 2.3 Experiment

The goal of this study is to clarify the impact of software licenses on CnP reuse. In our previous study [45], we performed a preliminary investigation on the impact on a small Java source file set. To confirm the findings obtained from the previous study, this paper presents two experiments on large-scale file set: counting CnPs according to licenses, and statistical examination of the impact.

The outline of the two experiments is as follows. First, the source files of OSS products are collected. Then, these source files are analyzed to detect instances of CnP. Finally, the instances are counted and assessed according to a certain criterion.

The following subsections describe the design and the procedure of detecting CnPs and the analyzed code. Moreover, the detail and the result of

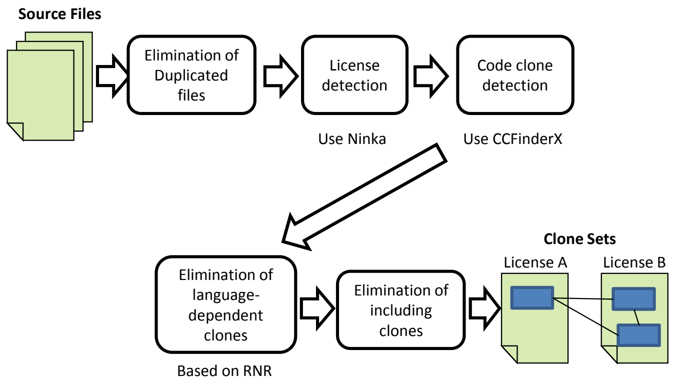


Figure 2.2: Overview of the process for detecting licenses and CnPs

the two experiments are also described.

### 2.3.1 Detecting CnP – Design and Implementation

To clarify the impact of software licenses on CnP reuse, the license of the files and CnPs in the files must be detected. We designed the detection process as Figure 2.2.

A large number of source files are used in the experiment, hence it is impractical to detect the licenses of all the files by hand. Therefore, we used Ninka [26] which is the automatic license detection tool for a source file. Note that Ninka can only detect the license of a source file when a known license description is included in the file.

If all activities of a developer were observed, all CnP could be extracted without any error. However, such observation is impossible since OSS developers are working worldwide. On the other hand, the history of the development is recorded in the repository used for OSS development. However, the intention or detailed activity unfortunately does not remain in the repository. Therefore, we decided to detect CnP instances only from snapshots of source code.

CCFinderX, a representative code clone detector, is one of publicly available tools for CnP detection. A code clone is a code fragment that is the same as or similar to another fragment. A code clone is classified into a clone set, which is an equivalence class of the clone relation. Code clones

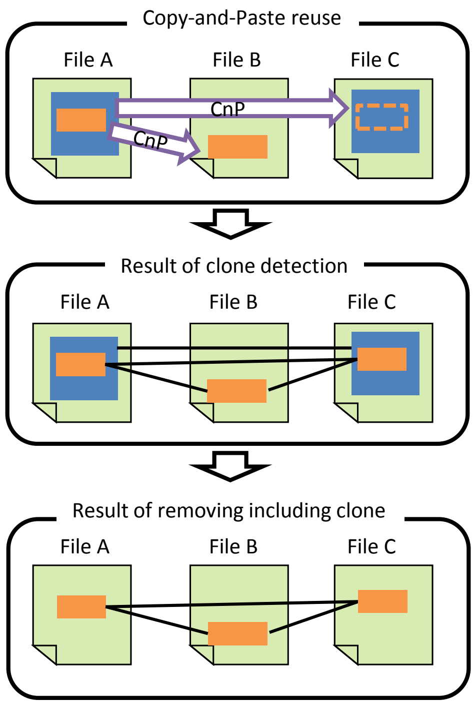


Figure 2.3: Removing including clones

are typically generated by CnP [49, 50, 51]. In previous work, code clones were also used for CnP detection [52, 53].

Using clones as CnP has two potential problems. One is that clones are not always suitable for counting CnPs. The other is that the direction of CnP is lost, since a clone set is a mathematical set. The loss of the direction is discussed at the end of this subsection.

Code clones which are not suitable for counting CnP as known as *language-dependent code clones* or *including code clones*. Examples of language-dependent code clones are consecutive if blocks, case entries of switch statements, and consecutive variable declarations. Language-dependent clones are usually generated not by CnP. The detail of language-dependent code clone is discussed in [54]. Including code clone is a code clone whose proper sub-fragment is another code clone. If including code clones are taken into counting, CnP is overcounted.

Figure 2.3 shows the overcounting and its solution. Suppose that a long code fragment is copied from file A to file C, and then a short code fragment in the long fragment is copied from file A to file B. Despite only 2 CnPs were occurred in the files, CCFinderX detects 5 pairs of code fragments as code clones. To avoid overcounting, code clones that include other code clones (including clones) are removed from the clone detection result. As a result of the removing, number of detected clones matches 1 origin and 2 copied fragments.

Detail of the five phases in the detection process (Figure 2.2) is described below.

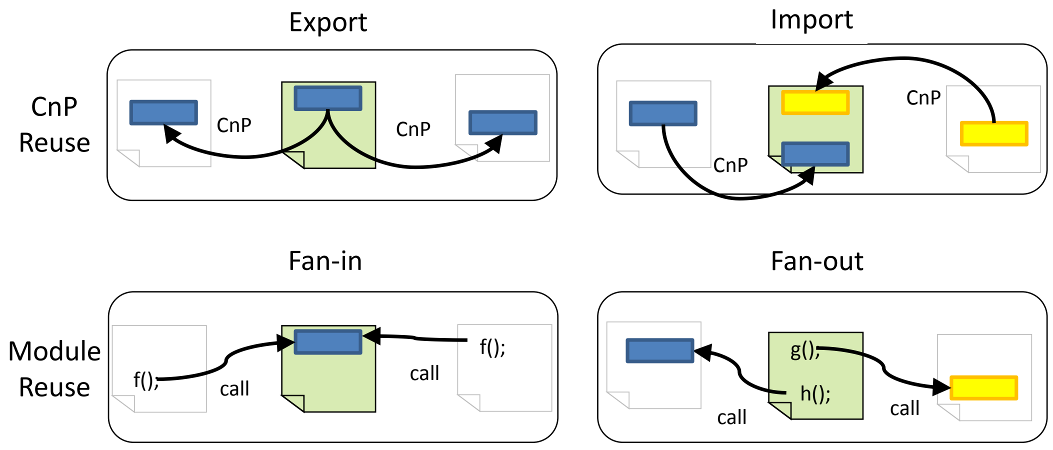


Figure 2.4: Similarity between CnP reuse and module reuse

1. **Elimination of Duplicated files:** Eliminate identical files in the target collection of source files except for one of them, because the experiment focuses on only copy-and-paste reuse of code fragments rather than reuse of entire files.

2. **License detection:** Identify the license of each file in the output of phase 1 using Ninka. The files which do not include license a description are eliminated and never passed to the next phase.

3. **Code clone detection:** Identify the location of code clones using CCFinderX from the collection of source files delivered from phase 2.

4. **Elimination of language-dependent clones:** To eliminate language-dependent code clones, eliminate code clones which RNR (Ratio of Non-Repeated tokens) metric [54] is higher than 0.5 in the code clones detected in phase 3.

5. **Elimination of including clones:** Eliminate overlapped code clones which include other clones from the input from phase 4.

### Discussion on treating code clones as CnPs

Due to substituting a clone set of the CnP instances, the direction of CnP is lost since a clone set is a mathematical set. Therefore, we have to consider the loss of direction may affect the experimental results.

In general, there are two cases that CnP increases clones in an external file: importing a code fragment from another file, or exporting a code fragment to another file. Import and export of code fragment respectively correspond to fan-in and fan-out in case of module-level reuse. (Figure 2.4). Ichii et al. revealed that the distribution of fan-out is far narrow compared to the distribution of fan-in, since the maximum value of fan-out is restricted by the file size [55]. The distributions of export and import are same as fan-in and fan-out.

This analogy shows that the number of clones which relate to a certain file is dominated by exporting, because the sum of the two distributions is basically determined by a broader one. Therefore, substituting a clone for a CnP should be enough for catching the trend of reusing.

### 2.3.2 Analyzed Code

Two sets of source files are analyzed in the experiments.

The first dataset (**DS1**) contains all the C/C++ files in packages of *main* section in Debian/GNU Linux 5.0.2 lenny [25]. DS1 contains 776,289 files (286MLOC) obtained from 6,472 packages. DS1 is composed of common and widely-used OSS products.

The second data set (**DS2**) contains C/C++ source files sampled from SourceForge.net [2]. The sampled files are contained in 1,070 packages selected randomly which are developed in C/C++ and whose subversion repository has 10 or more commits. Selecting only packages whose repository has 10 or more commit is to exclude packages which have been seldom developed. As a result, DS2 contains 425,830 files (121MLOC). DS2 is designed as being representative of all the OSS products in the world.

DS1 and DS2 contain 41 same packages. These same packages do not spoil independence of the two data sets, since the 41 packages are only 0.6% of DS1 and 4% of DS2.

From the obtained source files, the clones and licenses are detected. A brief summary of detection is shown in the Table 2.2. The product license is explained using abbreviations of the license name for simplicity. Table 2.1 shows the basic abbreviations of the main licenses. “v” and the number immediately after a basic abbreviation denotes the license version. Additionally, a plus (+) sign immediately after the version number means “or any later”. If a product is distributed under a composite license such as dual-license or exception-clause, multiple license names are concatenated with commas and used as the license name of the product.

Table 2.1: Representative abbreviations of license names

| **Abbreviation** | **Name** |
|---|---|
| Apache | Apache Public License |
| BSD3 | Original BSD minus advertisement clause |
| GPL | General Public License |
| LesserGPL | Lesser General Public License |
| LibraryGPL | Library General Public License |
| MX11 | MIT License/X11License |
| MPL | Mozilla Public License |
| subversion | Subversion License |

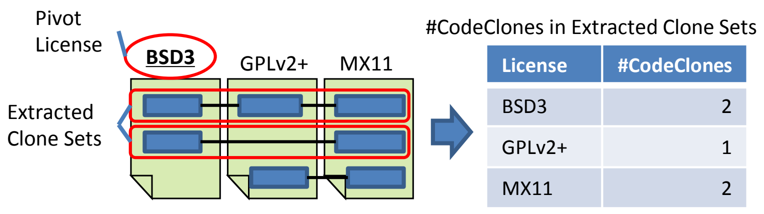
Table 2.2: Top 15 frequent licenses

(a) DS1

| | **License** | **#Files** |
|---|---|---|
| ✓ | GPLv2+ | 178,174 |
| | LibraryGPLv2+ | 28,000 |
| | LesserGPLv2.1+ | 24,540 |
| | GPLv2 | 22,840 |
| | GPLv3+ | 18,372 |
| | GPLv2or LGPLv2.1, MPLv1_1 | 15,897 |
| ✓ | BSD3 | 11,933 |
| ✓ | MX11 | 11,715 |
| | LesserGPLv2+ | 10,537 |
| | boostV1 | 9,275 |
| | GPLnoVersion | 5,354 |
| ✓ | Apachev2 | 4,297 |
| | LibraryGPLv2 | 4,187 |
| | BSD2 | 4,123 |
| | LesserGPLv2.1 | 3,709 |

(b) DS2

| | **License** | **#Files** |
|---|---|---|
| ✓ | GPLv2+ | 44,558 |
| | boostV1 | 13,461 |
| | GPLv3+ | 12,037 |
| | LesserGPLv2.1+ | 8,765 |
| | LibraryGPLv2+ | 6,705 |
| | GPLv2 | 6,674 |
| ✓ | Apachev2 | 6,220 |
| ✓ | BSD3 | 4,784 |
| | LesserGPLv2+ | 3,543 |
| | LesserGPLv2.1 | 2,943 |
| ✓ | MX11 | 2,478 |
| | BSD2 | 2,408 |
| | LesserGPLv3+ | 1,227 |
| | FreeType | 961 |
| | subversion+ | 868 |



#CodeClones in Extracted Clone Sets

| License | #CodeClones |
|---|---|
| BSD3 | 2 |
| GPLv2+ | 1 |
| MX11 | 2 |

Figure 2.5: Example of the Counting Code Clones in Experiment 1

### 2.3.3 Experiment 1: counting clones for each license

Experiment 1 counts the number of code clones grouped by their license for both data sets. Through probing the differences between the counts, we try to reveal the relationship between CnP reuse and software licenses.

#### Method

The following procedure is iterated until there are no further interesting licenses.

1. Manually select an arbitrary interesting license (the **pivot license**).

2. Extract the clone sets which include the source code under the pivot license.

3. Count the code clones in the extracted clone sets grouped by license (the **peripheral license**).

Figure 2.5 illustrates an example of this step. There are three files together with their licenses. Boxes in the files denote clones, and these clones are connected by lines to compose clone sets. Let us assume *BSD3* is the pivot license. Since the upper two clone sets include *BSD3* clones, these sets are extracted. Then, the clones in the sets are counted grouped by license.

#### Results

Distinguishing licenses are selected as pivot licenses from top popular licenses. (In Table 2.2, pivot licenses are marked on the left of the name).

Table 2.3: #Clones with pivot license in DS1

(a) with Apachev2

| **License** | **#Clones** |
|---|---:|
| GPLv2+ | 38,520 |
| Apachev2 | 15,618 |
| GPLv2 | 5,261 |
| LibraryGPLv2+ | 5,040 |
| LesserGPLv2.1+ | 4,491 |
| Others | 25,814 |

(b) with BSD3

| **License** | **#Clones** |
|---|---:|
| GPLv2+ | 94,301 |
| BSD3 | 66,271 |
| GPLv2 | 13,066 |
| LibraryGPLv2+ | 12,602 |
| LesserGPLv2.1+ | 12,269 |
| Others | 71,851 |

(c) with GPLv2+

| **License** | **#Clones** |
|---|---:|
| GPLv2+ | 794,569 |
| GPLv3+ | 114,476 |
| LibraryGPLv2+ | 70,804 |
| LesserGPLv2.1+ | 56,802 |
| GPLv2 | 50,658 |
| Others | 245,805 |

(d) with MX11

| **License** | **#Clones** |
|---|---:|
| GPLv2+ | 104,542 |
| MX11 | 84,482 |
| LibraryGPLv2+ | 13,830 |
| GPLv2orLGPLv2.1, MPLv1_1 | 13,738 |
| GPLv2 | 13,639 |
| Others | 79,950 |

Table 2.4: #Clones with pivot license in DS2

(a) with Apachev2

| License | #Clones |
|---|---:|
| Apachev2 | 9,336 |
| GPLv2+ | 5,903 |
| GPLv2 | 1,590 |
| GPLv3+ | 1,466 |
| LesserGPLv2.1+ | 1,016 |
| Others | 4,793 |

(b) with BSD3

| License | #Clones |
|---|---:|
| BSD3 | 8,686 |
| GPLv2+ | 7,394 |
| GPLv3+ | 1,667 |
| LibraryGPLv2+ | 1,479 |
| GPLv2 | 1,457 |
| Others | 7,002 |

(c) with GPLv2+

| License | #Clones |
|---|---:|
| GPLv2+ | 71,569 |
| GPLv3+ | 6,643 |
| LesserGPLv2.1+ | 6,143 |
| GPLv2 | 5,360 |
| LibraryGPLv2+ | 4,489 |
| Others | 24,418 |

(d) with MX11

| License | #Clones |
|---|---:|
| GPLv2+ | 6,985 |
| MX11 | 5,086 |
| LesserGPLv2.1+ | 1,276 |
| GPLv3+ | 1,136 |
| LibraryGPLv2+ | 1,128 |
| Others | 5,764 |

Table 2.5: Normalized values for #Clones (deeper colored cells depict high values)

(a) DS1

| | **Pivot License** | | | |
|---|---|---|---|---|
| | Apachev2 | BSD3 | GPLv2+ | MX11 |
| Apachev2 | 8.46E-04 | 3.30E-05 | 8.27E-06 | 3.72E-05 |
| BSD3 | 4.28E-05 | 4.65E-04 | 1.11E-05 | 5.13E-05 |
| GPLv2+ | 5.03E-05 | 4.44E-05 | 2.50E-05 | 5.01E-05 |
| MX11 | 5.91E-05 | 5.90E-05 | 1.42E-05 | 6.16E-04 |

(b) DS2

| | **Pivot License** | | | |
|---|---|---|---|---|
| | Apachev2 | BSD3 | GPLv2+ | MX11 |
| Apachev2 | 2.41E-04 | 3.23E-05 | 1.06E-05 | 4.00E-05 |
| BSD3 | 2.09E-05 | 3.80E-04 | 1.32E-05 | 3.88E-05 |
| GPLv2+ | 2.13E-05 | 3.47E-05 | 3.60E-05 | 6.33E-05 |
| MX11 | 1.64E-05 | 2.82E-05 | 1.49E-05 | 8.28E-04 |

Except the same type of license, four representative licenses are selected based on its popularity. Ultimately, pivot licenses are *GPLv2+*, *MX11*, *BSD3* and *Apachev2*.

Tables 2.3 and 2.4 show the counts for DS1 and DS2 with each pivot license. Each table shows peripheral licenses and the number of clones.

In all the resulting counts, the files distributed under *GPLv2+* contain most of the clones. In the result of DS1, the pivot license is generally second highest except in the case of *GPLv2+*. Similarly, in the result of DS2, the pivot license and *GPLv2+* rank at first and second in the all counting results.

Meanwhile, the number of source files strongly affects the number of clones. To exclude the affection, the number of clones is normalized by dividing by the number of pivot license files and peripheral license files. The normalized value means the expected number of the clones in a peripheral license file when there are only one pivot license file and one peripheral license file. Table 2.5 shows the normalized values. If the value is higher, the cell has deeper color.

Table 2.5 shows that the expected number of clones is highest when the peripheral license is the same as the pivot license in the case of *Apachev2*, *MX11* and *BSD3*.

Table 2.6 shows the number of clones per one pivot license file. These numbers suggest the tendency to be reused of each pivot license. The case of DS1 indicates that *MX11*, *BSD3* or *Apachev2* files tend to be reused frequently compared to *GPLv2+* files. Similarly, the case of DS2 indicates that *MX11*, *BSD3* files tend to be reused frequently compared to *GPLv2+*.

### 2.3.4 Experiment 2: statistical examination of licenses

The result of experiment 1 indicates the relationships between licenses and frequency of CnP reuse. For a more detailed investigation, we statistically examine the impact of the license on CnP count using the same data set.

#### Method

This experiment confirms whether the licenses affect to the number of CnP reuse even if the other factors which affect the reusability are removed. For the confirmation, two types of regression models are compared from the perspective of the fitness; One type of the models is only based on the factors which previous studies propose as reusability metrics, and the other type is based on both of license information and the reusability factors.

Table 2.6: Number of clones for one source file distributed under a pivot license

(a) in DS1

| | **#Clones** | **#Files** | **(#Clones) / (#Files)** |
|---|---:|---:|---:|
| Apachev2 | 94,744 | 4,297 | 22.04887 |
| BSD3 | 270,360 | 11,933 | 22.6565 |
| GPLv2+ | 1,333,114 | 178,174 | 7.482091 |
| MX11 | 310,181 | 11,715 | 26.47725 |

(b) in DS2

| | **#Clones** | **#Files** | **(#Clones) / (#Files)** |
|---|---:|---:|---:|
| Apachev2 | 24,104 | 6,220 | 3.875241 |
| BSD3 | 27,685 | 4,784 | 5.786998 |
| GPLv2+ | 118,622 | 44,558 | 2.662193 |
| MX11 | 21,375 | 2,478 | 8.625908 |

The response variable of the all regression models is the number of clones which relate to a file in a product and which are outside of the product. The counting rule is explained using the example in Figure 2.6. There is a clone set between three products. Assume the case counting the number of related clones focusing on leftmost file in product A. All clone sets that contains a clone in the focusing file are collected. The response variable is the number of clones in the collected clone sets, excluding the product that have the focusing file. The clones in the same product are out of count because CnP in a same product rarely causes a license problem.

Explanatory variables are licenses and the metrics which seem to relate to

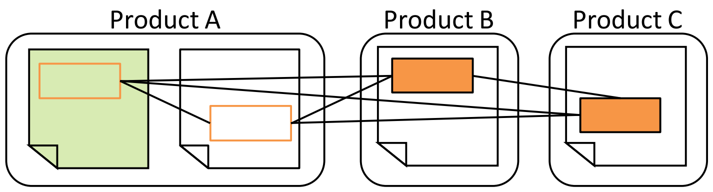


Figure 2.6: Counting extracted clones related to a file in a product

Table 2.7: The metrics for a file used in Experiment 2

| **Poulin's classification** | **Employed metrics** |
|---|---|
| Complexity | LOC: lines of code excluding comment<br>MCC: sum of McCabe's cyclomatic complexity |
| Documentation | COM: lines of comment |
| External dependencies | EXF: # of called functions defined in external<br>EXV: # of used variables defined in external |
| Proven reliability | AGE: elapsed seconds since the file was created |

reusability. Table 2.7 shows the 6 metrics employed as explanatory variables. The metrics are selected for covering Poulin's classification of reusability attributes [56]. Unfortunately, metric AGE is only applicable for DS2 since the repository of DS1 (Debian lenny) does not contain created time of the files.

Since license information is a nominal scale, license information must be transformed into *indicator variables*. Top 15 licenses shown in Table 2.2 are selected for the indicator variables for each data set.

Three regression models for each data set are made of the above mentioned variables. The regression models are described below. Both the response variable and the metric values are logarithmically converted.

$$\log(\text{number of related clones} + 1) =$$

$$\epsilon + \sum_{m \in M} \alpha_i \log(m+1) \tag{M1}$$

$$\epsilon + \sum_{m \in M} \alpha_i \log(m+1) + \sum_{l \in L} \beta_i l \tag{M2}$$

$$\epsilon + \sum_{m \in M} \alpha_i \log(m+1) + \sum_{l \in L} \beta_i l + \sum_{m \in M} \sum_{l \in L} \gamma_{ij} l \log(m+1) \tag{M3}$$

where $M$ is a set of metrics, and $L$ is a set of indicator variables of Licenses.

The last clause of M3 means the interactions between the metrics and licenses. If the licenses affect CnP frequency, models M2 and M3 fit the data better than M1. The result of regression analysis is identified by the combination of data set and model names, e.g. DS2-M3 means the result of model M3 using dataset DS2.

As described above, the models are not straightforward linear expressions but logarithmically converted. Straightforward linear models are discarded,

Table 2.8: Adjusted coefficient of determination values

(a) for DS1

| **Model** | $R^2$ |
|---|---|
| DS1-M1 | 0.5021 |
| DS1-M2 | 0.5047 |
| DS1-M3 | 0.5133 |

(b) for DS2

| **Model** | $R^2$ |
|---|---|
| DS2-M1 | 0.3396 |
| DS2-M2 | 0.3522 |
| DS2-M3 | 0.3692 |

because they show far low fitness compared to converted ones in our preliminary experiment. Generally, logarithmic conversion improves the fitness of regression models in many cases of software repository analysis.

Finally, fitness of the model is compared. The difference between residual of the models are verified using ANOVA. If the significant differences exist, *adjusted coefficients of determination* $R^2$ for the models are compared. If significant difference exists between two models, a model which has larger $R^2$ value fits to the data significantly better than another model.

### Result

First of all, differences between fitness of the models are tested by ANOVA. Tested pairs are <DS1-M1, DS1-M2>, <DS1-M2, DS1-M3>, <DS2-M1, DS2-M2>, and <DS2-M2, DS2-M3>. The comparison confirms that there are significant differences ($p$ <2.2e-16) in the all pairs.

Since difference of the fitness is confirmed, the degree of fitness can be compared by coefficients of determination $R^2$. Table 2.8 shows that the $R^2$ value for each model. The $R^2$ values increase in order of M1 to M3 for both data sets. This result confirms that the license of a file has a clear impact for CnP reuse, even if the impacts of other factors are eliminated.

### 2.3.5 Revisiting Research Questions

#### RQ1

The experiment 1 shows that *GPLv2+* code has been distinctly less often reused than code with other licenses. Also, this result is supported by the significant impact of the license confirmed in the experiment 2. According to those results, we can conclude that *the source code distributed under a permissive license is more frequently reused than that distributed under a restrictive license.*

### RQ2

In all investigations on the two data sets in the experiment 1, files distributed under a particular pivot license are most often imported into files distributed under the same pivot license. Furthermore, Table 2.5 shows that the probability of CnP is highest when a pivot and target licenses are the same, except for *GPLv2+*. Therefore, we can conclude that *source files that are distributed under a license are the most frequently imported into ones distributed under the same license.* On the other hand, *GPLv2+* was ranked as first or second in raw count, respectively. However, Table 2.5 shows that *GPLv2+* files less frequently import source code from other files. According to those results, we can conclude that *GPLv2+ files have a substantial impact on the reuse count because of available huge number of GPLv2+ files.*

## 2.4 Threats to Validity

To check the validity of the experiment and the conclusions, we discuss the threats to the validity from four viewpoints; i.e. external, internal, construct, and conclusion validity.

**External Validity** If a bias exists in the analyzed sets of source files, the generality of the experimental results will be compromised. This research uses two source file sets as analysis targets; source packages in Debian GNU/Linux 5.0.2 and OSS projects hosted at SourceForge.net. The set of source packages in Debian is a set of applications used frequently in the real world. On the other hand, SourceForge.net is a well-known site in the OSS community and registers various OSS products without intentional distinction. The experiment used source files randomly sampled from SourceForge.net. Therefore, the results should generalize to the entire OSS world since any bias in the analysis targets would be negligible.

**Internal Validity** The result of experiments may be incorrect if other factors that affect reuse count are missing in the model. Poulin [56] proposes that the factors that affect software reusability are classified into four categories: complexity, documentation, external dependencies, and reliability. Furthermore, all criteria proposed in the works on reusability measurement [57, 58] are also classified into the four categories. The metrics used in the experiment 2 covers the four factors, therefore the model should include sufficient factors.

**Construct Validity** If the number of reuse instances by CnP is not accurate, the experimental results may be incorrect. This research uses Ninka to identify the license of a source file. Ninka has high recall and precision [26]. Although files in which Ninka fails to identify the license are excluded from the data sets used in the experiment to detect CnP, this is conducted mechanically. Therefore, the impact on the experimental result would be small. Next, the experiment uses CCFinderX and a filter using the RNR metric [54] to identify reuse by CnP. CCFinderX and CCFinder, which is the previous version of CCFinderX, are used to detect CnP in several works [52, 53]. On the other hand, the filter using the RNR metric is used to remove code clones except by CnP in several works [59, 60]. Therefore, the usefulness of these tools has been shown experimentally. Note that we do not consider the direction of CnP in the experiments. Hence, the values from the experiment are larger than the actual values. However, the discussion of this experiment is based not on an absolute number of CnP instances, but on the difference in the numbers of CnP instances in this research. From the above, the numbers of CnP instances obtained in the experiment are adequate for comparing CnP instances in two different applications.

**Conclusion Validity** The validity of the method used to derive our conclusions should be discussed. The results of the experiment lead to two conclusions. Our first conclusion is that files under copyleft licenses such as GPL, are less frequently reused than files under other licenses. In the process of deriving this conclusion, four types of licenses, including one copyleft license, were examined in the experiment. This is fewer than the number of all OSS licenses. However, the analysis targets included many files under these target licenses and the two data sets used as the analysis targets showed common characteristics. In addition, the number of CnP instances in files under the copyleft license is very different from the other figures. Therefore, the difference between them is obvious. Our second conclusion is that files under the same license perform CnP more frequently than files under different licenses. This trend is visible in all the data sets and the licenses examined in the experiments. In addition, the number of CnP instances among files under the same license is ten times greater than the number of files under different licenses. From the above, the method for deriving these two conclusions has no major problems that could lead to incorrect conclusions.

## 2.5 Summary

This study shows the impact of software licenses on CnP reuse in C/C++ files. The results of the experiment show that CnP mostly occurs within source code distributed under the same license. On the other hand, a substantial amount of reused code fragments appears in the *GPLv2+* source code, since the number of *GPLv2+* files is very large. Furthermore, the results confirm that permissive licensed files tend to be reused more than copyleft ones. As a result, the features of CnP reuse expected by reason of license characteristic are confirmed quantitatively.

Chapter 3

# Building Domain Specific Dictionaries of Verb-Object Relation from Source Code

## 3.1 Introduction

Identifier names in source code are important in program comprehension which consumes a half of the time of program maintenance [61, 62]. A software developer generally tries to guess a role of a program element from identifier names [14, 15]. Therefore, identifiers should be named as its role in the program appropriately [63, 64]. If an identifier name is inappropriate, it is difficult for a developer to correspond a program element and a domain knowledge. Lawrie et al. [65] revealed that identifiers that are acronyms or meaningless serial numbers cause developers to waste much more time in program comprehension, compared to when identifiers are spelled out fully without any abbreviations.

Unfortunately, not all developers are able to give appropriate names to identifiers, since a broader knowledge and a great deal of experience are necessary to define accurate names. Developers need to learn the rules of various words and their combinations (e.g. naming rules) in different domains, such as programming language, development organization, and application domain. The only way to learn these rules is through examples, since the rules are not documented in many cases.

For the purpose of naming identifiers, our research group has built dictionaries containing good examples of identifier names. In a previous work, we proposed a method for building a dictionary of the super-sub (abstract-concrete) relation of nouns used in identifiers [28]. The dictionary seems to help naming of class names and variable names which consist of nouns.

However, identifier names in source code include not only nouns and their relationship, but includes verbs. In particular, a method name often includes a verb which represents the behavior of the method [63, 66, 67]. Additionally, an objective-phrase which represents the target of the method is often appeared behind a verb. The above dictionary include neither verb nor their relationships, so that it seems to not helpful for supporting naming of a method.

This research proposes a method for building a dictionary of verb-object (V-O) relations extracted from source code. In particular, verbs are extracted from a method name using existing natural language processing and the pattern matching system. Similarly, objects are extracted from the method name, names of formal parameters of the method, and the name of the class to which the method belongs. This process is applied to a set of source files categorized by application domain. Relations that appear frequently in a domain are included in the dictionary for that domain. The V-O dictionary seems to support the naming of methods [68].

In the evaluation, four domain dictionaries have been built from Java source file sets. The relationships in the dictionaries were evaluated participants who have Java program development experience and domain knowledge. As a result, it has been confirmed that most of the relationships are domain-specific relationships or relationships which is usual in Java programs.

The rest of the chapter is structured as follows. Section 3.2 explains in detail the V-O relationship in object-oriented programs and naming rules of methods, while Section 3.3 presents the algorithm used to build a dictionary of V-O relations from source code. Section 3.4 discusses the evaluation experiment and its results. Finally, Section 3.5 discusses related works, while Section 3.6 gives our conclusions and future works.

## 3.2 Verb-Object Relationships in Object-Oriented Program

### 3.2.1 Naming Convention

In general, it is recommended that an identifier has a specific and obvious name that expresses its role. Since a role of an identifier is sometimes complicated in object-oriented programs, an identifier in an object-oriented program is often expressed as a compound name, consisting of several words.

Unfortunately, white spaces are prohibited in identifiers in many programming languages, and therefore, alternative ways are employed to express compound names, i.e. camel case and snake case. Camel case refers to a concatenated string of words, whose first letters are capitalized (e.g. CamelCase). Snake case is a concatenated string of words with underscores between words (e.g. snake_case). In Java source code, camel case is recommended and popular [65].

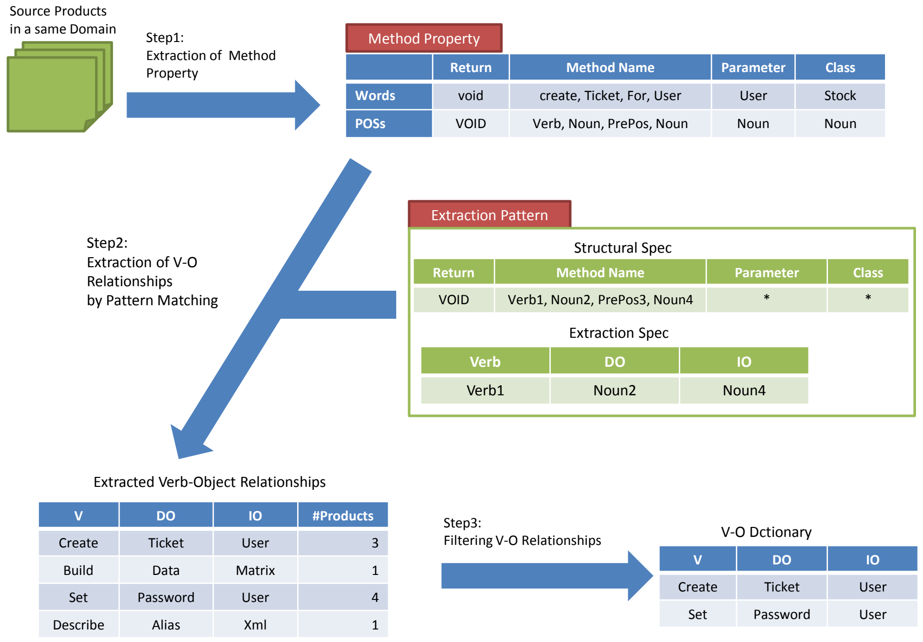
Commonly, the name of a method in an object-oriented program includes a verb. In most cases, the head of the method name is a verb or verb clause, followed by a noun or adjectives [66, 67]. In other cases, the head of the method name is a noun or adjective, or a noun or adjective clause, followed by a verb in the past tense. For example, `java.awt.event.ActionLister` in the Java API [69] has a method `actionPerformed(ActionEvent)`.

In a few cases, the method name contains no verbs. For example, the names of the `toString()` method of `java.lang.Object` and `newInstance()` method of `java.lang.Class` do not contain any verbs. From one point of view, these methods omit the obvious verbs, such as convert or create. The alternative view is that `to` and `new` act as verbs in the methods. This paper adopts the latter view.

### 3.2.2 Verb and Object in a Method Name

Object-oriented programs contain many statements describing operations on targets. An operation and a target correspond to a verb and an object, respectively. Receiver or parameter objects are used as the targets of the operation. Fry et al. [70] proposed a method to extract verbs and direct objects from method names, parameters, and class names.

In source code, there are many pairs of verbs and objects that seldom appear in natural language texts. For example, `java.net.Socket` has a method `bind(SocketAddress)`. The method name actually means “bind SocketAddress to Socket”. However, in natural language text, except for programming documents, socket almost never acts as an object of the verb “bind”. Moreover, the verb-object relations that appear in source code are sometimes specific to a certain program domain, since programs in different domains use different words, or the same words with a different meaning. For instance, in a database domain, noun `cursor` is used as a pointer to a selected data record for the purpose of accessing data records one-by-one. On the other hand, in the GUI application domain, cursor means an editing point of a text area. Only in a database domain does `fetch from cursor`


Source Products in a same Domain
Step1: Extraction of Method Property
Method Property

|  | Return | Method Name | Parameter | Class |
|---|---|---|---|---|
| Words | void | create, Ticket, For, User | User | Stock |
| POSs | VOID | Verb, Noun, PrePos, Noun | Noun | Noun |

Step2: Extraction of V-O Relationships by Pattern Matching
Extraction Pattern
Structural Spec

| Return | Method Name | Parameter | Class |
|---|---|---|---|
| VOID | Verb1, Noun2, PrePos3, Noun4 | * | * |

| Verb | DO | IO |
|---|---|---|
| Verb1 | Noun2 | Noun4 |

Extracted Verb-Object Relationships

| V | DO | IO | #Products |
|---|---|---|---|
| Create | Ticket | User | 3 |
| Build | Data | Matrix | 1 |
| Set | Password | User | 4 |
| Describe | Alias | Xml | 1 |

Step3: Filtering V-O Relationships
V-O Dctionary

| V | DO | IO |
|---|---|---|
| Create | Ticket | User |
| Set | Password | User |

Figure 3.1: Outline of our Technique

mean to acquire one set of data from the database.

## 3.3 Building Verb-Object Relationship Dictionary

This section describes a method to build a dictionary of V-O relations by extracting the relations from object-oriented programs. The input to the method is source files for a certain domain written in an object-oriented language, while the output is a dictionary composed of tuples consisting of a verb (**V**), direct-object (**DO**), and indirect-object (**IO**) specific to the domain. Note that the IO field may be empty. An outline of the approach is shown in Figure 3.1. The approach consists of the following three steps.

**Step 1: Obtaining the identifiers related to each method.** Retrieve all method declarations in the input source, and obtain identifiers that relate to declarations. Then, make a *method property* for each method declaration. A method property is a tuple which includes four elements representing an identifier and its attribute.

**Step 2: Extracting V-O relations.** Extract tuples consisting of V, DO, and IO from method properties using with pattern matching.

**Step 3: Filtering V-O relations.** From the output of step 2, select the tuples that appear in more than a certain number of software products.

The following subsections describe the detail of the steps.

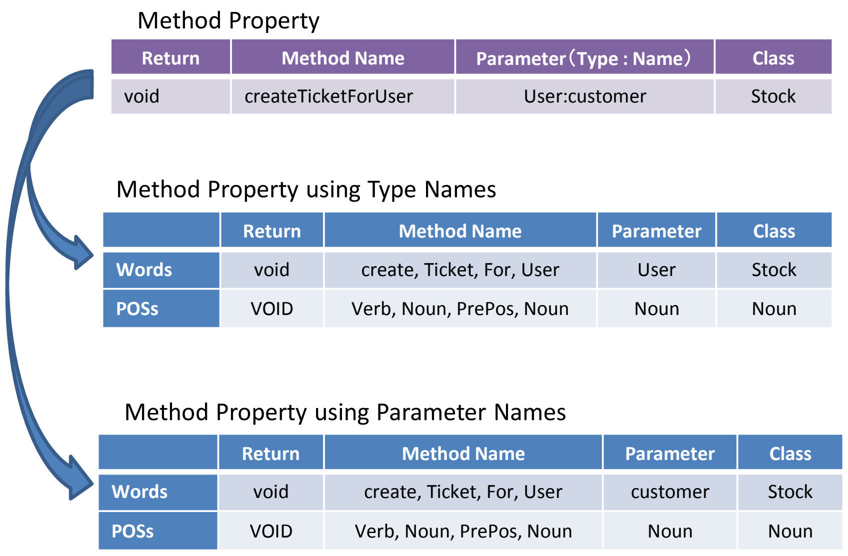
### 3.3.1 Step 1: Obtaining the identifiers related to each method

This step extracts all the method declarations and related identifiers from the input source files, and then makes method properties for each method.

A method property is a tuple of four sequences which include words together with their respective parts of speech. The four sequences correspond to the return type of the method, the name of the method, the parameters of the method, and the name of the class to which the method belongs, respectively. Since a parameter has two types of identifiers which include formal parameter names and type names of the formal parameter, two method properties are made from a method if the method has one or more parameters: one contains the sequence of types of the parameters in the third element, while the other contains the sequence of names of the parameters (as shown in Figure 3.2).

The procedure to acquire the method property from the identifiers obtained in the previous step is described below.

- An element corresponding to return type is a tuple of the name of the return type and its part-of-speech. If the return type is not `void`, the name of the return type is treated as a noun, otherwise, added as a special tag "VOID".
- An element corresponding to method name is a sequence derived from a method name. First, the method is parsed into a word sequence with camel case and snake case. Next, the part-of-speech tags are identified using OpenNLP [71].
- An element corresponding to a parameter is a sequence with length greater than 0. The $n$-th formal parameter of the method corresponds to the $n$-th element of the sequence. As described above, formal parameter names and their types are respectively used for two method properties. Each word in both is treated as a single noun word.



Method Property

| Return | Method Name | Parameter（Type : Name） | Class |
|---|---|---|---|
| void | createTicketForUser | User:customer | Stock |

Method Property using Type Names

| | Return | Method Name | Parameter | Class |
|---|---|---|---|---|
| Words | void | create, Ticket, For, User | User | Stock |
| POSs | VOID | Verb, Noun, PrePos, Noun | Noun | Noun |

Method Property using Parameter Names

| | Return | Method Name | Parameter | Class |
|---|---|---|---|---|
| Words | void | create, Ticket, For, User | customer | Stock |
| POSs | VOID | Verb, Noun, PrePos, Noun | Noun | Noun |

Figure 3.2: An example of Creating Method Properties

- An element corresponding to a class name is a sequence containing only the class name which is treated as a noun.

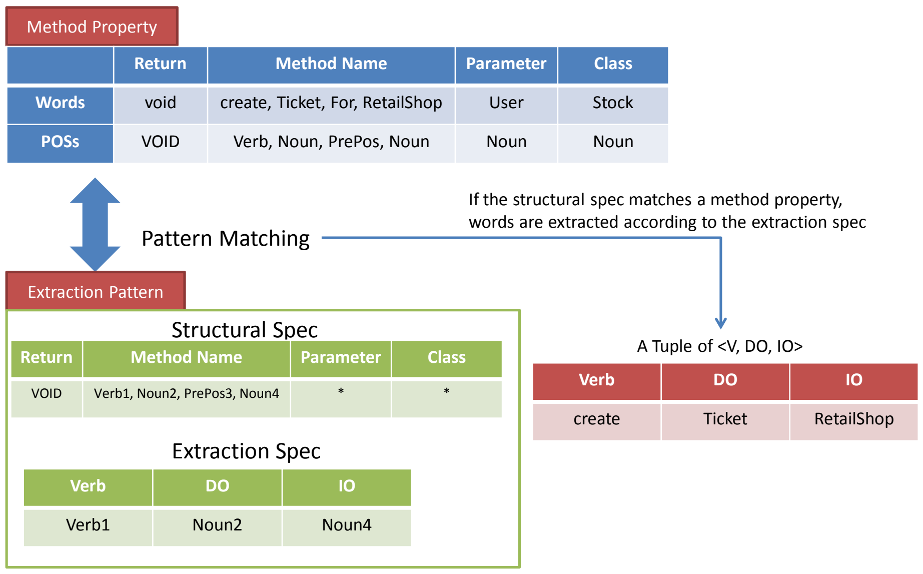
### 3.3.2 Step 2: Extracting V-O relations

Pattern matching is used to obtain a tuple (V, DO, IO) from the method property (see Figure 3.3). The pattern (**extraction pattern**) is composed of two parts, the **structure spec** and **extraction spec**.

The structure spec is represented by a tuple of four elements. These four elements correspond, respectively, to the return type, name of the method, parameters, and class name. Unlike a method property, the elements of the tuple are not sequences of words. An element of a structure spec is a wild card or a sequence of pairs consisting of a part-of-speech and a word number. Note that a wild card is also represented by $*$ in this chapter.

The pattern match extracts (V, DO, IO) tuples from a method property according to an extraction spec if its structure spec matches the method property. A structure spec matches the method property only when all of the following conditions are satisfied.

1. For each $n$-th element of the structure spec and the method property,


Method Property
Pattern Matching
Extraction Pattern
If the structural spec matches a method property, words are extracted according to the extraction spec
Structural Spec
Extraction Spec
A Tuple of <V, DO, IO>

| | Return | Method Name | Parameter | Class |
|---|---|---|---|---|
| Words | void | create, Ticket, For, RetailShop | User | Stock |
| POSs | VOID | Verb, Noun, PrePos, Noun | Noun | Noun |

| Return | Method Name | Parameter | Class |
|---|---|---|---|
| VOID | Verb1, Noun2, PrePos3, Noun4 | * | * |

| Verb | DO | IO |
|---|---|---|
| Verb1 | Noun2 | Noun4 |

| Verb | DO | IO |
|---|---|---|
| create | Ticket | RetailShop |

Figure 3.3: Method Property, Extraction Pattern, and Pattern Matching

the element of the structure spec is a wild card, or both sequences have strictly the same part-of-speech in the same order.

2. If the structure spec has the same word number at several points, the corresponding words in the method property are the same.

The extraction spec is represented by a tuple of three word numbers. The third number may be empty. If the structure spec matches a method property, the words that correspond to the word numbers specified in the extraction spec are extracted as V, DO, and IO words, respectively. If the third number is empty, IO is also empty.

The method properties obtained in the previous stage are collated with the patterns. If two or more patterns match a single method property, multiple tuples for each pattern are extracted from the single method property.

### 3.3.3 Step 3: Filtering V-O relations

This step filters the output of step 2, and builds a dictionary of V-O relations that frequently appear in the source code. In particular, only those tuples that appear in a certain number of software products are included in the dictionary. The threshold is tuned by hand.

## 3.4 Evaluation Experiment

We performed an experiment to evaluate the validity of the dictionary built using the proposed method. This section describes the experiment in detail, together with its result, and then examines and discusses the result.

### 3.4.1 Experimental Setup

We prepared 29 extraction patterns by hand which are shown in Table 3.1 Note that the leftmost column shows an ID number of each pattern. Extraction patterns have been prepared by the following steps. First, we assumed method signatures which are clearly identified (V, DO, IO) tuples, and then made basic extraction patterns for the method signatures, e.g. Pattern 20 for getter method, and pattern 8 for setter method. Next, we analyzed method signature which were not matched defined patterns, then made a pattern for the unmatched method signature. The analysis and making repeated until all method were matched at least one pattern.

Table 3.1: List of the Extraction Patterns (V, N, PP mean verb, noun and prepositional word, respectively)

| | Structure Spec | | | | Extraction Spec | | |
|---|---|---|---|---|---|---|---|
| ID | Return Type | Method Name | Parameters | Class Name | V | DO | IO |
| 1 | * | V1 N2 PP3 N4 | * | * | V1 | N2 | N4 |
| 2 | * | V1 PP2 N3 | * | N4 | V1 | N4 | N3 |
| 3 | * | V1 N2 | * | N3 | V1 | N2 | N3 |
| 4 | * | V1 PP2 N3 | N4 | * | V1 | N4 | N3 |
| 5 | * | V1 N2 PP3 | N4 | * | V1 | N2 | N4 |
| 6 | void | V1 | (empty) | N2 | V1 | N2 | (empty) |
| 7 | void | V1 PP2 N3 | (empty) | N4 | V1 | N4 | N3 |
| 8 | void | V1 N2 | N2 | N3 | V1 | N2 | N3 |
| 9 | void | V1 N2 PP3 N4 | * | N5 | V1 | N2 | N5 |
| 10 | void | V1 | N2 | N3 | V1 | N2 | N3 |
| 11 | void | V1 N2 | N3 | N4 | V1 | N3 | N4 |
| 12 | void | V1 N2 | N3 | N2 | V1 | N2 | N3 |
| 13 | void | V1 N2 | (empty) | N2 | V1 | N2 | (empty) |
| 14 | void | N1 V2 | N3 | N4 | V2 | N1 | N3 |
| 15 | void | N1 V2 | N1 | N3 | V2 | N1 | N3 |
| 16 | N1 | V2 N1 | N3 | N1 | V2 | N1 | N3 |
| 17 | N1 | V2 N1 PP3 N4 | N4 | N5 | V2 | N1 | N4 |
| 18 | N1 | V2 N3 PP4 N5 | (empty) | N6 | V2 | N3 | N5 |
| 19 | N1 | V2 PP3 N4 | N5 | N6 | V2 | N6 | N4 |
| 20 | N1 | V2 N1 | (empty) | N3 | V2 | N1 | N3 |
| 21 | N1 | V2 | N3 | N4 | V2 | N4 | N3 |
| 22 | N1 | V2 PP3 | N4 | N5 | V2 | N5 | N4 |
| 23 | N1 | V2 PP3 N4 | * | * | V2 | N1 | N4 |
| 24 | N1 | V2 PP3 N4 | (empty) | N1 | V2 | N1 | N4 |
| 25 | N1 | V2 PP3 | N4 | N4 | V2 | N4 | N4 |
| 26 | N1 | V2 N3 | (empty) | N1 | V2 | N3 | N1 |
| 27 | N1 | V2 | (empty) | N3 | V2 | N3 | (empty) |
| 28 | N1 | V2 N3 | (empty) | N3 | V2 | N3 | N1 |
| 29 | N1 | V2 N1 | (empty) | N1 | V2 | N1 | (empty) |

The target of extraction was source code for 38 open source software products shown in Table 3.2. The products were classified into four domains,Web Applications, XML Processing, Databases, and GUIs, which are abbreviated as Web, XML, DB, and GUI, respectively. For each domain, source files were analyzed and a dictionary built.

### 3.4.2 Evaluation Target Dictionaries

Table 3.3 gives the output of the analysis before filtering. Since several methods produce two or more tuples, the number of tuples is greater than the number of methods that match any patterns.

Table 3.4 shows the frequency distribution of tuples obtained from a certain number of software products. This table shows that the number

Table 3.2: Applications for Building Dictionaries

| | |
|---|---|
| **Web Applications** | |
| BBS-CS 8.0.3 | JForum 2.1.8 |
| JGossip 1.1.0.005 | mvnForum 1.2.1 |
| Yazd Discussion Forum Software 3.0 | Order Portal 1.2.4 |
| Arianne RPG 0.80 | JBoss Wiki Beta2 |
| JSP Wiki 2.8.3 | SnipSnap 1.0b3 |
| **XML** | |
| Castor 1.3 | DOM4J 1.6.1 |
| JDOM 1.1.1 | Piccolo 1.04 |
| Saxon-HE 9.2.0.5 | Xalan-J 2.7.1 |
| Xbeans 2.0.0 | Xerces-J 2.9.0 |
| XOM 1.2.4 | XPP3 1.1.4 |
| xstream-1.3.1 | |
| **Databases** | |
| Axion 1.0 Milestone 2 | Apache Derby 10.5.3 |
| H2 1.2.128 | HSQLDB 1.8.1.1 |
| Berkeley DB Java Edition 4.0.92 | Mckoi 1.0.3 |
| MyOODB 4.0.0 | NeoDatis 1.9.22.674 |
| OZONE 1.1 | tinySQL 2.26 |
| **GUIs** | |
| ArgoUML 0.28.1 | BlueJ 2.5.3 |
| Eclipse Classic 3.5.1 | jEdit 4.3.1 |
| NetBeans 6.8 | vuze 4.3.1.2 |
| LimeWire 5.4 | |

of tuples produced by two or more applications is significantly decreased against the number of tuples produced by only one application.

Based on the above results, we set two as the threshold for filtering, i.e. tuples appearing in two or more software products were included in the dictionaries. We think that if the threshold is set at three, the number of tuples is too small to evaluate.

Table 3.5 shows the number of methods which are successfully matched with each pattern. A digit in parentheses is a ratio of the methods against all methods in the source file set. The table shows that the extraction spec 3 takes tuples from more than half of the methods. Similarly, the extraction spec 11 takes many tuples, then to the extraction spec 3.

Table 3.3: The number of Methods and Extracted Tuples

| | # of methods | # of methods matched at least one method | Ratio of the matched methods | # of tuples |
|---|---:|---:|---:|---:|
| Web | 74,707 | 67,276 | 90% | 67,429 |
| XML | 55,812 | 46,885 | 84% | 49,926 |
| DB | 74,127 | 60,326 | 81% | 63,087 |
| GUI | 298,696 | 247,918 | 83% | 273,202 |

Table 3.4: Frequently Distribution of Tuples

| | # of products producing tuples | | | | | |
|---|---:|---:|---:|---:|---:|---:|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| Web | 67,147 | 258 | 18 | 4 | 2 | 0 |
| XML | 49,379 | 465 | 63 | 13 | 5 | 1 |
| DB | 62,415 | 609 | 28 | 1 | 32 | 2 |
| GUI | 272,795 | 339 | 38 | 23 | 5 | 2 |

Table 3.5: The number of Methods Extracted Tuples by Extraction Pattern

| ID | DB | GUI | WEB | XML |
|---|---|---|---|---|
| 1 | 2782 (3.75) | 7535 (2.52) | 4188 (5.61) | 1714 (3.07) |
| 2 | 1612 (2.17) | 4300 (1.44) | 1165 (1.56) | 757 (1.36) |
| 3 | 49951 (67.39) | 201859 (67.58) | 59015 (79.00) | 39540 (70.84) |
| 4 | 492 (0.66) | 1826 (0.61) | 482 (0.65) | 327 (0.59) |
| 5 | 114 (0.15) | 875 (0.29) | 126 (0.17) | 124 (0.22) |
| 6 | 2048 (2.76) | 9217 (3.09) | 798 (1.07) | 1091 (1.95) |
| 7 | 309 (0.42) | 561 (0.19) | 346 (0.46) | 92 (0.16) |
| 8 | 2072 (2.80) | 11752 (3.93) | 4994 (6.68) | 3679 (6.59) |
| 9 | 1423 (1.92) | 2882 (0.96) | 2891 (3.87) | 962 (1.72) |
| 10 | 1218 (1.64) | 6372 (2.13) | 607 (0.81) | 1089 (1.95) |
| 11 | 5865 (7.91) | 34064 (11.40) | 4803 (6.43) | 5384 (9.65) |
| 12 | 18 (0.02) | 47 (0.02) | 8 (0.01) | 11 (0.02) |
| 13 | 34 (0.05) | 38 (0.01) | 9 (0.01) | 8 (0.01) |
| 14 | 474 (0.64) | 10826 (3.62) | 347 (0.46) | 230 (0.41) |
| 15 | 17 (0.02) | 1092 (0.37) | 10 (0.01) | 3 (0.01) |
| 16 | 16 (0.02) | 35 (0.01) | 7 (0.01) | 28 (0.05) |
| 17 | 27 (0.04) | 101 (0.03) | 98 (0.13) | 15 (0.03) |
| 18 | 587 (0.79) | 1651 (0.55) | 363 (0.49) | 166 (0.30) |
| 19 | 152 (0.21) | 719 (0.24) | 158 (0.21) | 225 (0.40) |
| 20 | 2829 (3.82) | 7310 (2.45) | 1278 (1.71) | 2461 (4.41) |
| 21 | 1239 (1.67) | 6358 (2.13) | 523 (0.70) | 1363 (2.44) |
| 22 | 85 (0.11) | 408 (0.14) | 17 (0.02) | 92 (0.16) |
| 23 | 604 (0.81) | 2233 (0.75) | 549 (0.73) | 493 (0.88) |
| 24 | 2 (0.00) | 7 (0.00) | 0 (0.00) | 1 (0.00) |
| 25 | 23 (0.03) | 43 (0.01) | 4 (0.01) | 14 (0.03) |
| 26 | 139 (0.19) | 849 (0.28) | 67 (0.09) | 211 (0.38) |
| 27 | 774 (1.04) | 1886 (0.63) | 367 (0.49) | 814 (1.46) |
| 28 | 22 (0.03) | 81 (0.03) | 14 (0.02) | 28 (0.05) |
| 29 | 7 (0.01) | 22 (0.01) | 0 (0.00) | 10 (0.02) |

### 3.4.3 Evaluation Process

The resulting dictionaries were evaluated by 6 students in a software engineering laboratory. The participants all had experience in software development in Java. Moreover, they evaluated the dictionaries for the domains in which they had some experience.

For each domain dictionary, the tuples (V, DO, IO) were evaluated from the following perspectives.

**Perspective 1** The V-O relation of the tuple is actually used in the domain or in Java programs.

**Perspective 2** The verb, direct-object, and indirect-object are suitable.

**Perspective 3** The tuple is useful for appropriate naming of identifiers.

Based on the perspectives, we prepared several questions for the participants. The following three questions were based on the first perspective.

**Q1** Is this (V, DO, IO) tuple popular in the domain of the dictionary?

**Q2** Is this (V, DO, IO) tuple popular in common Java programs?

**Q3** Is this (V, DO, IO) tuple popular in another domain? If so, give the domain.

The following question was prepared according to the second perspective.

**Q4** V, DO and IO are correctly extracted? If you do not think so, identify the incorrect words.

The following three questions were based on the third perspective.

**Q5** Should this (V, DO, IO) tuple be given as a good example to a developer who is naming an identifier in a program in this domain?

**Q6** Should this (V, DO, IO) tuple be given as a good example to a developer who is naming an identifier in a common Java program?

**Q7** Should this (V, DO, IO) tuple be given as a good example to a developer who is naming an identifier in a program in another domain? If so, give the domain.

Table 3.6: Response to Q1

| | (A) | (B) | (C) | (D) | (Z) |
|---|---|---|---|---|---|
| Web | 21 | 35 | 7 | 3 | 24 |
| XML | 44 | 18 | 5 | 7 | 16 |
| DB | 32 | 36 | 4 | 9 | 9 |
| GUI | 42 | 26 | 9 | 6 | 7 |

Table 3.7: Response to Q2

| | (A) | (B) | (C) | (D) | (Z) |
|---|---|---|---|---|---|
| Web | 16 | 29 | 10 | 30 | 5 |
| XML | 15 | 11 | 17 | 42 | 5 |
| DB | 22 | 13 | 32 | 17 | 6 |
| GUI | 32 | 37 | 9 | 6 | 6 |

The participants selected answers for Q1, Q2, Q5 and Q6 from (A) strongly agree, (B) agree, (C) disagree, (D) strongly disagree, or (Z) no idea.

Each of the dictionaries was evaluated by two participants, and each of the participants evaluated two different dictionaries. 15 tuples that appeared in three or more software products and 15 tuples that appeared in two software products were evaluated by the participants. The tuples were randomly and exclusively selected from the dictionary. However, since the dictionary for the web application domain only contains 24 tuples appearing in three or more products, only 6 tuples were evaluated by two participants.

### 3.4.4 Results of the Evaluation

First, we describe the results obtained according to perspective 1. Table 3.6 gives the results for Q1, and Table 3.7 the results for Q2. The sum of the percentages of (A) strongly agree and (B) agree is 62% to 75% for Q1, and 38% to 76% for Q2. These results show that the greater part of the dictionaries consist of domain specific relations. However, several domain independent relations are also included.

Table 3.8 gives the responses for Q3. All dictionaries contain some V-O relations from another domain. The dictionary could be improved by separating these relations into another dictionary. Table 3.9 shows the example

Table 3.8: Response to Q3(Numbers in parentheses mean the number of the same answers)

| | |
|---|---|
| Web | Database(16), I/O processing(6), Common Java Programs(2) |
| XML | Data Analysis(2), GUI(1), Parser(1), Resource Management(1), Tree Structure(1), Graph Processing(1) |
| DB | GUI(5), Web Application(1), String Processing(1) |
| GUI | Database(1), Networking(1), Program Test Cases(1), Archiver(1), Common Java Programs(4) |

Table 3.9: Tuples being popular in other domains at responses of Q3

| | Verb | Direct Object | Indirect Object | Answered domain |
|---|---|---|---|---|
| Web | Set | Password | User | Database |
| XML | Perform | Action | ActionEvent | GUI |
| DB | Release | Mouse | MouseEvent | GUI |
| GUI | Open | Connection | Handler | Database, Networking |

Table 3.10: Response to Q4

| | Verb | DO | IO | Two or more |
|---|---|---|---|---|
| Web | 3 | 1 | 3 | 6 |
| XML | 5 | 7 | 1 | 12 |
| DB | 1 | 6 | 5 | 11 |
| GUI | 8 | 1 | 0 | 9 |

Table 3.11: Tuples answered as incorrect at responses of Q4

| | V | DO | IO | Inappropriate point |
|---|---|---|---|---|
| Web | Page | Exists | WikiEngine | V, IO |
| XML | Start | Dtd | XmlWriter | DO |
| DB | Perform | Action | Evt | IO |
| GUI | To | String | Mode | V |

Table 3.12: Response to Q5

| | (A) | (B) | (C) | (D) | (Z) |
|---|---|---|---|---|---|
| Web | 19 | 32 | 11 | 4 | 24 |
| XML | 33 | 15 | 10 | 16 | 16 |
| DB | 35 | 29 | 10 | 10 | 6 |
| GUI | 28 | 30 | 13 | 11 | 8 |

Table 3.13: Response to Q6

| | (A) | (B) | (C) | (D) | (Z) |
|---|---|---|---|---|---|
| Web | 13 | 19 | 23 | 30 | 5 |
| XML | 14 | 13 | 12 | 46 | 5 |
| DB | 31 | 13 | 24 | 16 | 6 |
| GUI | 29 | 26 | 15 | 13 | 7 |

Table 3.14: Tuples evaluated Useful at the Target Domain

| | Verb | DO | IO |
|---|---|---|---|
| Web | Destroy | Session | HttpSessionEvent |
| XML | Declare | Prefix | NamespaceSupport |
| DB | Add | Constraint | Table |
| GUI | Click | Mouse | MouseEvent |

Table 3.15: Response to Q7 (Numbers in the parentheses mean the number of the same answers)

| | |
|---|---|
| XML | Data Analysis(1), GUI(1), Parser(1)C<br>Resource Management(1), Tree Structure(1), Graph Processing(1) |
| DB | GUI(5), Web Application(1) |

Table 3.16: Extraction patterns and responses of tuples (numbers in each cell correspond to the responses of Q1/Q2/Q5/Q6)

| ID | A | B | C | D | Z |
|---|---|---|---|---|---|
| 1 | 7/3/9/4 | 17/3/13/3 | 1/15/5/12 | 2/10/3/12 | 7/3/4/3 |
| 2 | 1/0/1/0 | 0/2/0/1 | 1/0/1/1 | 0/0/0/0 | 1/1/1/1 |
| 3 | 78/49/65/52 | 66/45/58/35 | 11/36/20/36 | 15/62/27/69 | 34/12/34/12 |
| 4 | 5/3/3/3 | 0/1/1/1 | 0/0/1/0 | 0/1/0/1 | 0/0/0/0 |
| 5 | 2/1/1/1 | 2/3/1/2 | 0/0/2/1 | 1/3/1/3 | 2/0/2/0 |
| 6 | 8/4/7/6 | 5/6/6/4 | 2/4/2/4 | 0/2/0/2 | 1/0/1/0 |
| 7 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 8 | 14/10/12/10 | 12/9/11/7 | 2/6/5/8 | 2/7/2/7 | 3/1/3/1 |
| 9 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 10 | 1/0/0/0 | 1/0/2/0 | 0/1/0/0 | 0/3/0/4 | 4/2/4/2 |
| 11 | 9/8/9/7 | 7/4/6/4 | 3/2/3/3 | 1/8/2/8 | 2/0/2/0 |
| 12 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 13 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 14 | 30/13/22/12 | 16/25/20/18 | 3/9/4/16 | 3/7/5/7 | 4/2/5/3 |
| 15 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 16 | 0/1/0/1 | 1/0/1/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 17 | 1/0/1/1 | 1/0/1/0 | 0/2/0/0 | 0/0/0/1 | 0/0/0/0 |
| 18 | 4/3/5/3 | 10/0/9/0 | 1/11/4/10 | 2/6/2/7 | 5/2/2/2 |
| 19 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 20 | 7/10/6/8 | 9/7/3/3 | 2/1/4/1 | 3/5/8/11 | 2/0/2/0 |
| 21 | 3/3/3/3 | 1/3/1/3 | 2/0/2/0 | 0/0/0/0 | 2/2/2/2 |
| 22 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 23 | 1/3/1/2 | 3/2/1/3 | 1/1/3/1 | 3/3/3/3 | 2/1/2/1 |
| 24 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 25 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 26 | 0/0/0/0 | 1/0/1/0 | 0/1/0/0 | 0/0/0/1 | 0/0/0/0 |
| 27 | 1/2/1/1 | 0/1/0/2 | 2/0/2/0 | 0/1/0/1 | 2/1/2/1 |
| 28 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |
| 29 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 | 0/0/0/0 |

of such tuples. As shown in this table, (Release, Mouse, MouseEvent) is included in the DB dictionary, although the tuple belongs to the GUI domain.

Next, we give the results in perspective 2. Table 3.11 gives the responses for Q4. The percentage of tuples with incorrect V, DO, or IO values vary between 6% and 13%. This indicates that the 29 extraction patterns we prepared could be improved. Table 3.11 shows the example. As shown in the examples of DB and XML, tuples used acronyms such as Dvt and Evt were judged as incorrect. In addition, as shown in the example of GUI, "To" which we have decided to identify as a verb is judged as incorrect.

Finally, we present the results in perspective 3. Tables 3.12 and 3.13 give the results for Q5 and Q6, respectively. The concrete examples evaluated as useful are shown in Table 3.14. The sum of the percentages for (A) and (B) is between 53% and 71% for Q5, and between 30% and 61% for Q6. The results for Q5 and Q1 indicate that we need not only to improve precision of the dictionaries, but also to select and provide the relations that developers prefer. On the other hand, the results for Q6 show that the dictionaries contain many tuples that are suitable as naming examples for common Java programs. These tuples should be separated into a domain independent dictionary.

Table 3.15 gives the responses to Q7. Similar to the results for Q3, the responses for Q7 confirm the need to separate these tuples into another dictionary.

Moreover, we evaluated what extraction pattern contributed the useful tuples. Table 3.16 shows the number of responses for tuples (Q1/Q2/Q5/Q6) grouped by extraction pattern which takes the tuple. This table shows that the extraction pattern 3 most contributed for taking useful patterns. In addition, the table also shows that the extraction patterns 14 and 20 taking many useful tuples, although the number of methods corresponded to these patterns were relatively small.

### 3.4.5 Follow-up clarification of the results

Several tuples were determined to be undesirable to developers, despite the tuples are popular either in the domain or in common Java programs. We asked the participants to give reasons for this. The answers are given below.

- The tuple contains uncertain words. (e.g. abbreviation)
- The tuple is common sense for average developers.
- The tuple is used not in the whole domain, but in programs that are dependent on a specific library.

### 3.4.6 Discussion

The results for Q2 and Q3 show that the dictionaries contain V-O relations in another domain. The contamination might be as a result of the following two reasons.

- The threshold for filtering is too low to remove noise.
- Since software products generally span several domains, the relations in the domains covered by each of the input products, also span several domains.

The first problem is easy to solve by increasing the number of input products.

Regarding the second problem, we have a solution as described below. First, increase the number of domain categories and input products, and then classify the products into the domain categories on a nonexclusive basis. Thereafter, we track the original products of the tuples. If a tuple is a candidate of two or more dictionaries, the best dictionary to include the tuple is selected based on the original products and their domains.

### 3.4.7 Threats to Validity

From the viewpoint of the dictionary, the data assignment is performed randomly, since the tuples are randomly extracted from the dictionaries. However, the input software products were collected intentionally, and the number of products may be insufficient.

Regarding the participants, they are students in a graduate school, and not professional software developers. However, since they have experience of software development through part-time jobs or research projects, they have sufficient knowledge of several target domains to evaluate the dictionaries. Since the number of participants was not large enough, we could not randomly assign participants to dictionaries.

## 3.5 Related Work

This section introduces related works on V-O relations in the source code, and the differences between these works and our approach.

Shepherd et al. [72] and Fry et al. [70] extracted V-O relations from methods or comments in source code written in an object-oriented language, and used the relations for feature location and aspect mining. Hill et al. [73]

extracted the (V, DO, IO) tuples from source code and applied the tuples to feature location and aspect mining. Our approach is to build domain-specific dictionaries suitable for providing identifier names.

Host et al. built a dictionary of verbs from the name and body of methods [74], and created several naming rules from the dictionary to detect and fix incorrect naming [75]. Our approach is more relaxed compared to that in [75]; many good examples can be obtained using a verb with an associated object.

## 3.6 Summary and Future Work

This paper proposed an approach for building a domain specific dictionary of verb-object relations. The entries in the dictionary, i.e. tuples consisting of (V, DO, IO), are extracted from identifiers related to a method. For the evaluation experiment, 29 extraction patterns were made manually, and then four domain dictionaries were made from 38 software source codes. The dictionaries were evaluated six subjects. As a result, we confirmed that the entries in the dictionaries are popular in the domain, or common in programs written in Java.

One of the future work is about the extraction patterns. We made the 29 extraction patterns, but these have not been sufficient for any applications to extract all V-O patterns. Therefore, future work includes developing a meta algorithm of making extraction patterns, or developing a probability model which is independent on the particular patterns.

In the experiment, we did not evaluate the diversity of the dictionaries. Since common Java source code includes a lot of *getter* and *setter* methods, most of tuples in our dictionaries may be these methods. However, the dictionary should include various relationships, so that future work includes evaluating the diversity. In addition, developing an approach for extracting various V-O relationships is also included in future work.

In the proposed approach, a domain of software using as input is identified manually. However, to input a lot of software, these domains should be identified automatically. Therefore, future work includes using an automatic domain identification method and evaluating the output dictionary. We also aim to improve the precision of the dictionaries and to build larger dictionaries from a larger set of source code files.

# Chapter 4

# Comparison of Backward Slicing Techniques for Java

## 4.1 Introduction

Program slicing [18] is an analysis technique developers use to extract statements related to a specific behavior of interest. In particular, program slicing extracts a set of statements that may affect the value of a variable in a developer-specified statement. The set of statements extracted from a program is called a program slice.

Existing studies have reported the effectiveness of program slicing for supporting reuse activity. Lanubile et al. [19] proposed an approach for making a new reusable component from existing software based on program slicing [18]. Similarly, Komondoor [20] and Marx et al.[21] proposed approaches for component extraction based on program slicing, respectively. The effectiveness of program slicing in other fields has also been reported. Kusumoto et al. [30] report that program slicing is effective for debugging tasks. If a variable has an incorrect value, developers can use the respective program slice to investigate program statements that are likely to have output the incorrect value. In addition to debugging, program slicing has been adopted by several advanced analysis techniques, e.g. information-flow analysis [76] and change impact analysis [77].

Program slices should be accurate, in order to ensure that developers can concentrate on the smallest number of statements during their tasks. The System Dependence Graph (SDG) [33] has been proposed to represent how statements interact with one another in a program to compute a program slice. A program slice is obtained by backward traversal on an SDG from a vertex corresponding to a variable in a specified statement. Although SDG cannot determine the minimal program slice [78], several techniques have been proposed to approach it. For example, Allen et al. propose representing exception handling in SDG [37]. SDG has also been extended to represent Java language constructs [34] and data-flow via the fields of objects [35, 36, 24]. The accuracy of SDG data-flow information depends on the underlying points-to analysis accuracy, e.g. set-based [79, 80], object sensitive [81], and hybrid context sensitive analysis [82].

Today, not only accuracy, but also scalability is important. Acharya et al. [77] combine a lightweight program slicing technique with an accurate one for change impact analysis, because the accurate technique is very time consuming. Studies [22, 23] propose Static Execute Before/After analysis to alternate program slicing. The analysis depends on only a control-flow graph, instead of a traditional SDG. Consequently, Static Execute Before/After is lightweight and scalable, though less accurate than an SDG-based technique. Beszedes et al. [23] report that the technique is useful for impact analysis.

Tailoring program slicing for developers' and researchers' needs is an important issue. Java is the most popular programming language, in both open source [38] and industrial software development [39]; however, accuracy and scalability of Java program slicing techniques have not yet been investigated. Binkley et al. [40] evaluate program slicing for C/C++. They compare slices obtained with various configurations of CodeSurfer [41]. Jasz et al. [22] compare static execute before analysis and program slicing for C/C++. Beszedes et al. [23] compare static execute after analysis with forward program slicing in C/C++ and Java. They did not include a simple backward program slicing technique and static execute before analysis for Java, in the comparison. Moreover, improved slicing [24], an advanced slicing technique for Java, has not been evaluated with practical applications.

One of the key differences of Java and C/C++ is method parameters. Java only supports call-by-value parameter, while C/C++ has both call-by-value and call-by-reference parameter. As a result, a parameter cannot be used as output directly in Java. Instead of call-by-reference parameters, fields of objects in a parameter are often used. Therefore, in Java analysis, treatment of a field is more important than that in C/C++. The other key difference is a treatment of virtual method call. A method call in Java is implicitly treated as a virtual method call, while a virtual method call in C/C++ is used only if the called method is declared as virtual method. Consequently, conservative analysis to resolve frequent virtual method calls may increase the size of program slices. Furthermore, Java includes several dynamic features such as reflection. These language differences may cause the difference between the slicing results of Java and C/C++.

We compared slicing techniques through Java program analysis, to answer the following research questions.

**RQ1.** How accurate and scalable are slicing techniques compared to each other?

**RQ2.** Which slicing technique is most appropriate in specific situations?

Our analysis compared four slicing techniques as follows.

**Static Execute Before (SEB) (See Section 4.2.1 for details):** A lightweight technique based on control-flow analysis. Given a program statement, SEB extracts statements that may be executed before an execution of the statement which is completed. This approach is proposed as a replacement of program slicing [22]. Although SEB is context sensitive and never lost dependences, this technique has potential incorrectness caused by ignoring data dependences.

**Context-insensitive Slicing (CIS) (Section 4.2.2):** A simple program slicing technique [40], based on control-dependence and data-dependence analysis with an SDG which data dependence edges directly connect vertices representing field read/write statements instead of making trees of fields. CIS extracts statements that may affect the execution of a given statement. CIS covers all possible data-flow paths, and therefore may include infeasible control-flow paths.

**Intersection of SEB and CIS (HYB) (Section 4.2.3):** An improvement of the two aforementioned techniques by combining these techniques. HYB extracts an intersection of the statements extracted by both SEB and CIS, and excludes infeasible paths from CIS. This is natural improvement, but is not evaluated. Therefore, we introduced this technique for comparison.

**Improved Slicing (IMP) (Section 4.2.4):** A sophisticated program slicing technique for Java [24]. This is an extended version of traditional program slicing technique [33] in order to precisely analyze Java. This technique analyzes the data structure of objects and how objects are manipulated in feasible control-flow paths. It is expected to extract a more accurate program slice than the other techniques. However, it is also expected that the analysis cost is higher than the others. The actual precision and scalability of IMP in Java is unclear.

Note that our comparison focuses on strategies to represent a program as a graph, rather than the accuracy of the underlying analysis techniques, such as points-to analysis, because all four methods can use the same analysis techniques as infrastructure.

We apply the four program slicing techniques to six applications in DaCapo Benchmarks [83], in the comparison. We investigated scalability with two configurations: *Application separate from the library*, and *the whole system including the library.* We analyzed only an application with the former configuration, by approximating control-flow and data-flow information in the library. We analyzed all the control-flow and data-flow paths in both the application and the library with the latter configuration.

In addition, we investigated the scalability of IMP in detail since IMP could not analyze the whole system in the above experiment. We prepared various analysis configurations using the six applications and sub packages in java and javax packages. We measured the analysis time and the analyzability by performing SDG construction of the configurations.

The rest of the paper is organized as follows. Section 4.2 presents the concepts of the four program slicing techniques. Section 4.3 explains the implementation details of our slicing tool. Section 4.4 describes the experiment using the tool. Sections 4.5 and 4.6 discuss the results and the threats to validity, respectively. Section 4.7 explains related work. Finally, Section 4.8 summarizes the study.

## 4.2 Slicing Techniques Under Evaluation

This section introduces the basic ideas of the four slicing techniques compared in this study. All four techniques extract a program slice in two steps: *Graph Construction*, and *Graph Traversal.* Each technique constructs a graph representation of a target program, in the graph construction step. A slice for a given program element is extracted by graph traversal, in the graph traversal step. After a graph is constructed once, all slices can be extracted from that graph.

Tables 4.1 and 4.2 show examples used in the following subsections to explain the differences among the techniques. Both of their source code column show the same example program. The main method of the program is located on line 2. It calls four methods `init`, `pass`, `sum`, and `mult` in sequential order. Two of the methods, `sum` and `mult`, call the same method `foo`. Note that the `init` method creates values for the `sum` and `mult` methods. The `pass` method does nothing for the other methods.

Table 4.1: Example of Source Code and its Slicing Results (The criteria is y at line 5)

| Line | Source code | **SEB** | **CIS** | **HYB** | **IMP** |
|---|---|---|---|---|---|
| 1 | `class Main {` | | | | |
| 2 | `  public static void main(String[] args) {` | | | | |
| 3 | `    A a = init();` | ✓ | ✓ | ✓ | ✓ |
| 4 | `    pass();` | ✓ | | | |
| 5 | `    int y = sum(a);` | ✓ | ✓ | ✓ | ✓ |
| 6 | `    int z = mult(a); }` | | ✓ | | |
| 7 | `  static A init() {` | | | | |
| 8 | `    A a = new A();` | ✓ | ✓ | ✓ | ✓ |
| 9 | `    a.x = 1;` | ✓ | ✓ | ✓ | ✓ |
| 10 | `    return a; }` | ✓ | ✓ | ✓ | ✓ |
| 11 | `  static int sum(A a) {` | | | | |
| 12 | `    int l = a.foo();` | ✓ | ✓ | ✓ | ✓ |
| 13 | `    return 1 + l; }` | ✓ | ✓ | ✓ | ✓ |
| 14 | `  static int mult(A a) {` | | | | |
| 15 | `    int l = a.foo();` | | ✓ | | |
| 16 | `    return 10 * l; }` | | ✓ | | |
| 17 | `  static void pass() { return ; }` | ✓ | | | |
| 18 | `}` | | | | |
| 19 | `  class A {` | | | | |
| 20 | `    int x;` | | | | |
| 21 | `    int foo() { return this.x; } }` | ✓ | ✓ | ✓ | ✓ |

Tables 4.1 and 4.2 show not only source code, but also results of slicing. The criterion of Tables 4.1 and 4.2 are the variable `y` in line 5, and the variable `z` in line 6, respectively. The right columns of the both tables show the slicing results by each slicing technique for comparison in the following subsections. The top row of the columns shows the slicing technique. Each cell shows whether the slice includes the corresponding line or not. If a cell includes a check mark, the slice includes the corresponding line. For example, the slice of SEB with respect to the slicing criterion `y` includes lines 3, 4, and 5 in `main`, and all lines in `init`, `pass`, `sum`, and `foo`.

### 4.2.1 SEB: Static Execute Before

SEB [22] extracts statements that may be executed before the statement of interest execution is completed. SEB uses a control-flow graph for each method and a call graph for a target application. Given a program statement in a method, SEB identifies call sites that may directly or transitively

Table 4.2: Example of Source Code and its Slicing Results (The criteria is `z` at line 6)

| Line | Source code | **SEB** | **CIS** | **HYB** | **IMP** |
|---|---|---|---|---|---|
| 1 | `class Main {` | | | | |
| 2 | `  public static void main(String[] args) {` | | | | |
| 3 | `    A a = init();` | ✓ | ✓ | ✓ | ✓ |
| 4 | `    pass();` | ✓ | | | |
| 5 | `    int y = sum(a);` | ✓ | ✓ | ✓ | |
| 6 | `    int z = mult(a); }` | ✓ | ✓ | ✓ | ✓ |
| 7 | `  static A init() {` | | | | |
| 8 | `    A a = new A();` | ✓ | ✓ | ✓ | ✓ |
| 9 | `    a.x = 1;` | ✓ | ✓ | ✓ | ✓ |
| 10 | `    return a; }` | ✓ | ✓ | ✓ | ✓ |
| 11 | `  static int sum(A a) {` | | | | |
| 12 | `    int l = a.foo();` | ✓ | ✓ | ✓ | |
| 13 | `    return 1 + l; }` | ✓ | ✓ | ✓ | |
| 14 | `  static int mult(A a) {` | | | | |
| 15 | `    int l = a.foo();` | ✓ | ✓ | ✓ | ✓ |
| 16 | `    return 10 * l; }` | ✓ | ✓ | ✓ | ✓ |
| 17 | `  static void pass() { return ; }` | ✓ | | | |
| 18 | `}` | | | | |
| 19 | `  class A {` | | | | |
| 20 | `    int x;` | | | | |
| 21 | `    int foo() { return this.x; } }` | ✓ | ✓ | ✓ | ✓ |

invoke the method that includes the statement. Then, SEB identifies other statements, using graph traversal on control-flow graphs. SEB also identifies statements in methods that may be invoked before the given method. SEB extracts statements that may affect a given statement. Hence, it can be seen as the most lightweight variant of program slicing.

For example, if `y` in the line 5 is selected to compute a slice, the result includes lines 3, 4, 5 and also lines with the methods `init`, `pass`, `sum`, and `foo`, as shown in Table 4.1. The methods `sum` and `foo` are included because the variable `y` on line 5 is returned from `sum`, and the method calls `foo`. While `foo` is called from both `sum` and `mult`, `mult` is not included in the result, because the call site is not executed before line 5.

### 4.2.2 CIS: Context-Insensitive Slicing

CIS extracts statements that may affect the execution of a given statement. CIS is based on the context insensitive slicing technique implemented in CodeSurfer [41] which is evaluated in the study of Binkely et.al. [40]. Similar to traditional program slicing, CIS makes SDG. The vertices represent statements in the program, and edges represent control dependence and data dependence in the program. CIS extracts a program slice with backward traversal from vertices that correspond to a selected statement.

Control dependence represents the effect of control statements (e.g. `if` statements), while control-flow represents only the order of execution sequence. Data dependence represents the def-use relationship of variables. SDG represents data flow relationships with vertices that represent formal parameters of the method and actual arguments of the method call. Parameter-in/out edges connect vertices for parameters, according to method call relationships. If a method call instruction is a virtual method call, firstly, a possibly callable methods from the call instruction are extracted using with a points-to analysis result. Next, call edges and parameter-in/out edges are drawn between the call instruction and all callable methods.

We extend SDG of the context insensitive slicing technique [41], to represent data dependence of object fields and class variables, to compute a program slice for Java. An SDG has a data dependence edge for a field access between statements $s$ and $t$, if $s$ writes a field of an object and $t$ may read the field of the object. Points-to analysis is used to check whether the two statements may have accessed the same object or not. Similarly, a data dependence edge exists between statements $s$ and $t$ if $s$ writes a class variable and $t$ reads the class variable.

Note that these data dependence edges for fields and class variables are potentially context-insensitive because these edges ignore method call relationships. As a result, performing context-insensitive slice rather than context-sensitive slice is required for keeping soundness of the slicing result.

Figure 4.1 is an excerpt of an SDG representing the program in Tables 4.1 and 4.2. A vertex with a label that is a method name represents a method entry vertex. A vertex with a label that is a digit represents a statement. A rectangle vertex, with a label that is a parameter name, represents a formal parameter. A data dependence edge, from line 9 to line 21, shows data dependence through field `x`. If a developer selects variable `y` on line 5, CIS would extract lines 3, 5, and 6 in the `main` method, and lines in the `sum`, `foo`, `init`, and `mult` methods. This is because vertices corresponding to those lines are reachable from the vertex corresponding to line 5. As shown in Tables 4.1, compared to the SEB of `y`, CIS excludes the line 4 and `pass` which are executed before line 5 but do not have data/control dependence on line 5.

One of the shortcomings of CIS is that it may include infeasible control-

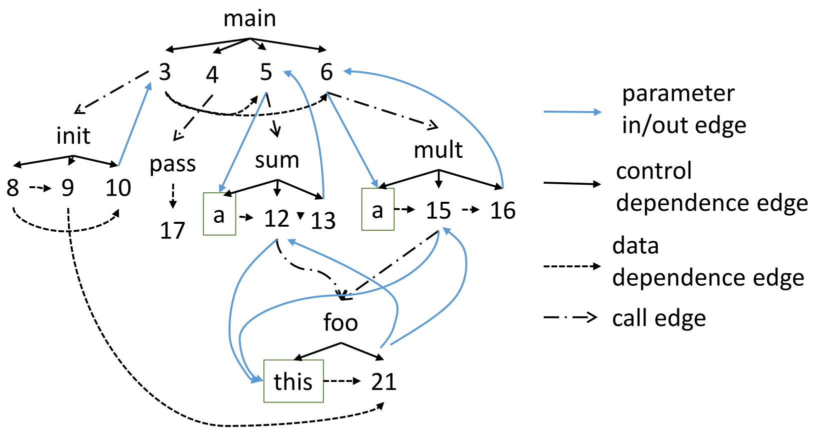


Figure 4.1: SDG for CIS

flow paths. An inter-procedural path is feasible if a method call in the path corresponds to a method return in the path. A control-flow path through lines 12, 21, and 12 is feasible, because it starts from a method call and correctly returns to the call site. On the other hand, a path through lines 15, 21, and 12 is infeasible, because it goes to another call site. Including `mult` in the slice of CIS for `y` would be caused by traversing this infeasible path.

### 4.2.3 HYB: Context-Insensitive Slicing with Static Execution Before

HYB extracts an intersection of statements obtained by SEB and CIS. As mentioned above, CIS for `y` includes `mult` as a false positive, even though `mult` is never called before line 5. HYB combines SEB and CIS to remove such infeasible statements from the result of CIS, by computing an intersection of the SEB and CIS results.

As shown in Table 4.1, a slice for `y` in line 5 includes lines of `sum`, `foo`, and `init`, while it excludes pass and `mult`. The result is just an intersection of the result of SEB for `y` and that of CIS. As a result, the result of HYB is improved by either of the two techniques.

### 4.2.4 IMP: Improved Slicing

IMP [24] extracts statements that may affect the execution of a given statement, similar to CIS. For C/C++, Liang et al.[35] extended traditional program slicing [33] to handle the fields of objects. IMP is a further extension of field handling for precise analysis of Java. IMP regards all fields accessed by a method as arguments of the method, instead of edges directly connecting field access between methods. This representation reflects how fields are manipulated through control-flow paths. Similar to CIS, if a method call instruction is a virtual method call, call edges and parameter-in/out edges are drawn between call instruction and all callable methods.

IMP represents fields of a parameter as a tree. Let `r.f` is a receiver object `r` and a field `f`, `p(r)` is points-to set of `r`. A vertex representing field `f` is added as a child of a vertex representing `r` if a vertex representing `r'` which points-to set `p(r')` is equal to `p(r)` does not exist in the pass from root to `r`. As a result, a tree made by the IMP's approach has at least one vertex representing an accessed field. Additionally, the IMP's approach can handle recursive structure distinguishing their points-to sets. Whereas the existing technique [35] stops expansion of a field tree at a specific level given by a parameter in order to avoid infinite expansion of a recursive data structure, IMP uses points-to information to stop infinite expansion.

A program slice is extracted by backward two-phase slicing [33], which avoids inter-procedural infeasible paths. For two-phase slicing, summary edges are prepared. A summary edge connects an actual-in vertex to an actual-out vertex if the actual-out vertex depends on the actual-in vertex. After building summary edges, two-phase slicing is performed. The first phase of two-phase slicing perform backward traversal from a given criteria along data dependence edges, control dependence edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. The second phase performs backward traversal from all actual-out vertices visited in the first phase along data dependence edges, control dependence edges, summary edges, and parameter-out edges, but not along call edges and parameter-in edges.

Figure 4.2 shows SDG for IMP, for the source code in Tables 4.1 and 4.2, with omitted vertices that represent actual arguments and summary edges [84]. Vertices that represent parameters not only have argument/parameter variable vertices, but also have field vertices. For example, vertex 10, which is the statement vertex and also formal-out vertex of `init`, has field vertex `x`, because the variable a returned in line 10 has a field `x`. Similarly, formal-in vertices of `sum`, `mult`, and `foo` have `x` as field vertices.

Figure 4.3 shows the subgraph of SDG for line 5 and `sum` including omitted vertices in Figure 4.2. Line 5 has the call instruction of `sum` which has argument `a` and returned value assigning to `y`. SDG in Figure 4.3 has vertices of `a` and `$Actual-out` which correspond to the argument and the

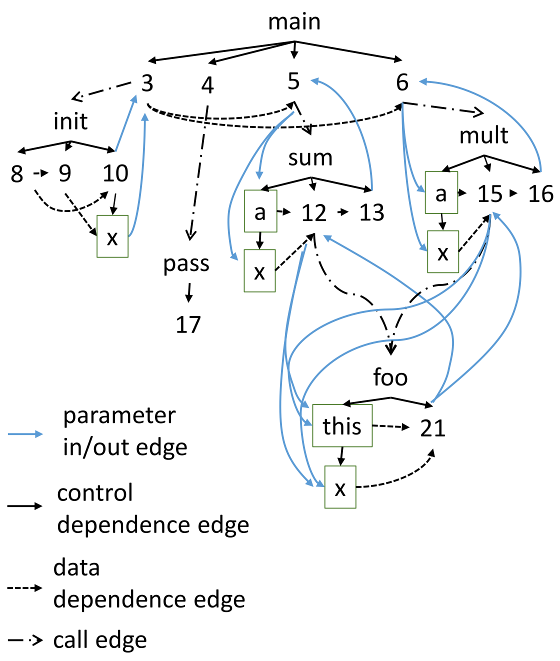


Figure 4.2: SDG for IMP

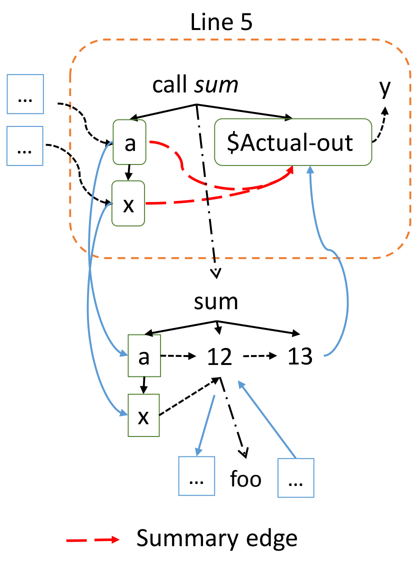


Figure 4.3: Subgraph of SDG for line 5

returned value respectively. Additionally, the SDG has a vertex `x` which represents a field `x` of the argument `a`.

A red dashed line in Figure 4.3 shows a summary edge. In an execution of `sum`, parameter `a` is used as a receiver object of call `foo` in line 12, and then a field `x` of `a` is used for the return value of `foo`. The returned value of `foo` in line 12 is used for return value of `sum` in line 13. As a result, argument `a` and `x` has transitive dependencies for return value of `sum`. Therefore, summary edges connect a vertex `a` and a vertex `x` to a `$Actual-out` vertex. Similar to the subgraph for line 5, subgraphs for lines 3, 12, and 15 have actual-in/out vertices and summary edges.

If the variable `y` on line 5 is selected, IMP performs two-phase slicing as follows: Firstly, first phase visits actual-out vertex of call `sum` via data dependence edge, and then actual-in vertices of call `sum` via summary edges, and more. As a result, first phase visits vertices of line 5 and line 3 including actual-in/out vertices. Second phase starts from actual-out vertices of line 5 and 3, and visits `sum`, `foo`, `init` via any edges except call and parameter-in edges. As a result, IMP extracts line 3, line 5, `sum`, `foo`, and `init`. Note that this result is the same as the result of HYB as shown in Table 4.1. HYB is effective in the case that SEB can remove infeasible paths from the result of CIS.

On the contrary, if the variable `z` on line 6 is selected, IMP extracts line 3, line 6, and `mult`, `foo`, and `init`. Table 4.2 shows that this is more precise than HYB, because HYB extracts `sum` in addition to the three other methods, because `sum` is executed before `mult`, and `sum` is reachable from `mult`, via an infeasible path in SDG.

### 4.2.5 Comparison of Graph Construction Process

Those four slicing techniques call both graph construction and control-flow analysis. SEB computation requires only the results of these analyses. CIS, HYB, and IMP all require additional processes, which include points-to and control/data dependence analyses. After that, SDG for CIS and HYB is constructed by making a traditional SDG and the connecting field/class variable. On the other hand, SDG for IMP is constructed by building a field tree of each argument and parameter, making the vertices and edges like in traditional SDG, and then computing the summary edges [84].

## 4.3 Implementation

We implemented four slicing techniques that target Java bytecode, because the bytecode is easier to analyze than the source code. Moreover, there are tools for points-to analysis and handling reflection targeting bytecode. We use the same points-to analysis and reflection handling process with all four slicing techniques. Therefore, all of the techniques under comparison use the same call graph and points-to information.

### 4.3.1 Points-to Analysis and Call Graph Construction

We used the Spark pointer analysis toolkit [85] with the Soot framework [86] for points-to analysis and call graph construction. Spark is implemented based on Andersen's points-to analysis algorithm [80], which is flow-insensitive, context-insensitive analysis. This context insensitive algorithm is good enough [87], because the slicing techniques under comparison do not require explicit context sensitivity, such as object sensitivity [81]. We used the default Spark configuration, the so called "on-fly-cg" and "field-sensitive", though Spark has many other options.

### 4.3.2 Handling Reflection

Reflection, which is a dynamic feature of Java, is the cause of imprecision in static program analysis. We used TamiFlex [88] to get the reflection result from a program execution. This tool can replace a reflection method invocation with a concrete method invocation, observed during a target program execution.

### 4.3.3 Approximation of Library

A slicing tool should analyze all methods reachable from the `main` method of the program, including the library used by the program, when computing a program slice. On the other hand, libraries may consume time and memory space for analysis, because class libraries, such as the JDK Platform API, include a large number of classes, even though most of them are not used by target programs. Additionally, some library methods are unanalyzable, e.g. native methods and methods protected by software license. We used the following approximations, to handle such unanalyzable code.

First, we assumed that for each library the method calls, the return value depends on the arguments. We connected edges from vertices representing arguments to their corresponding vertex representing the return value.

Secondly, we assumed that for collection classes (e.g. `List` and `Map`), method calls that modify a collection affect method calls that refer to the collection content. We manually listed methods such as `add` and `put` to modify a collection. We regarded other methods as those that refer to content. We translated the former method calls into statements, writing an artificial field representing collection, and the latter method calls into statements reading the field.

Finally, we excluded the `hashCode` and `equals` methods of all classes from analysis, if collection classes were not included in the analysis. This is because those methods are usually called back from collection classes.

## 4.4 Experiment

### 4.4.1 Design and Analysis Target

The goal of this experiment was to evaluate and compare scalability and accuracy of four slicing techniques. We measured the time required to construct an SDG and the size of the SDG, to analyze scalability. We measured the ratio of instructions included in a slice against the instructions in a target program, to analyze accuracy.

We analyzed six applications in DaCapo Benchmarks (version 9.12) [83]. DaCapo Benchmarks is a collection of real Java applications, with their execution scenarios. The six applications we used are avrora, batik, h2, luindex, pmd, and sunflow. Table 4.3 shows the size of the applications based on the number of methods that are reachable from their `main` methods. The table shows the number of classes, which includes reachable methods, the number of reachable methods, and the number of bytecode instructions in reachable methods. The columns "APP" and "LIB" show the numbers of classes/methods/instructions for application classes and library classes, respectively. We regard classes in the following packages as library classes: *java.**, *javax.**, *sun.**, *com.sun.**, *com.ibm.**, *org.xml.**, *apple.awt.**, and *com.apple.**. We obtained this list of packages from a TamiFlex document [89].

First, we executed the *default* application execution scenarios with TamiFlex, to record invoked methods by reflection. Next, Soot performed points-to analysis and call graph construction, with the output of TamiFlex. We constructed an SDG based on this information. A slicing criterion is a vertex in the SDG that corresponds to a bytecode instruction in APP. We performed backward slicing with each slicing criteria, and then measured the size of each slice.

Table 4.3: Size of Analysis Targets

| | #Classes | | #Methods | | #Instructions | |
|---|---|---|---|---|---|---|
| | APP | LIB | APP | LIB | APP | LIB |
| avrora | 49 | 1,701 | 290 | 10,697 | 9,690 | 321,666 |
| batik | 146 | 4,133 | 674 | 26,459 | 26,336 | 769,825 |
| h2 | 130 | 1,741 | 670 | 11,118 | 18,875 | 331,844 |
| luindex | 74 | 1,705 | 380 | 10,710 | 11,678 | 322,652 |
| pmd | 139 | 1,712 | 480 | 10,762 | 13,838 | 322,857 |
| sunflow | 58 | 3,751 | 331 | 24,144 | 11,786 | 684,263 |

We compared the relative slice sizes against an application, because the four techniques define graphs differently. We computed slice size, as the ratio of the number of application method instructions in the slice to the number of application method instructions in the program.

We used two different configurations. The first was *Application separated from library*, which analyzes application classes with library approximation. The second was *the whole system including library* that analyzes the whole application and its libraries without approximation.

We used a machine with Windows 7 64bit, Intel Xeon E5-2620 2.00GHz 2 processor CPU, and 64GB of RAM.

### 4.4.2 Result

#### Configuration 1: Application separated from library

Figure 4.4 shows the bar charts of the time required to construct an SDG for each application. Note that HYB used the same SDG as CIS, so that those times are the same. Table 4.4 shows the time of each step. The columns "Points-to Analysis (all)" and "Control-flow Analysis (all)" show the time required for points-to analysis and control-flow analysis. These analyses are common for all techniques. The column "Dependence Analysis (CIS/HYB)" shows the time required to analyze dependencies for CIS and HYB, using the information obtained by the points-to and control-flow analyses. The column "Dependence Analysis (IMP)" shows the time required to analyze dependencies for IMP, using the same information.

The time required for SDG construction was dependent on the time required for points-to and control-flow analyses, in this configuration. CIS took only a few additional seconds, while dependence analysis of IMP took

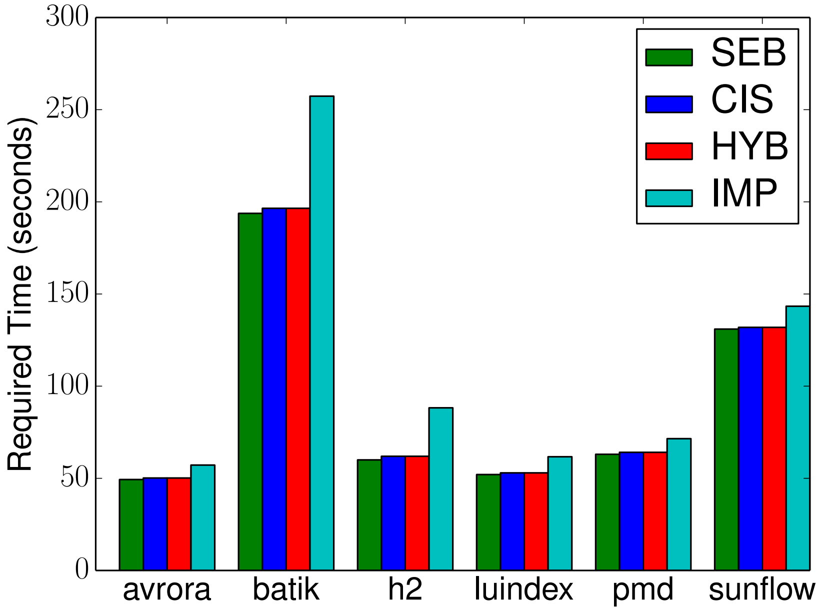


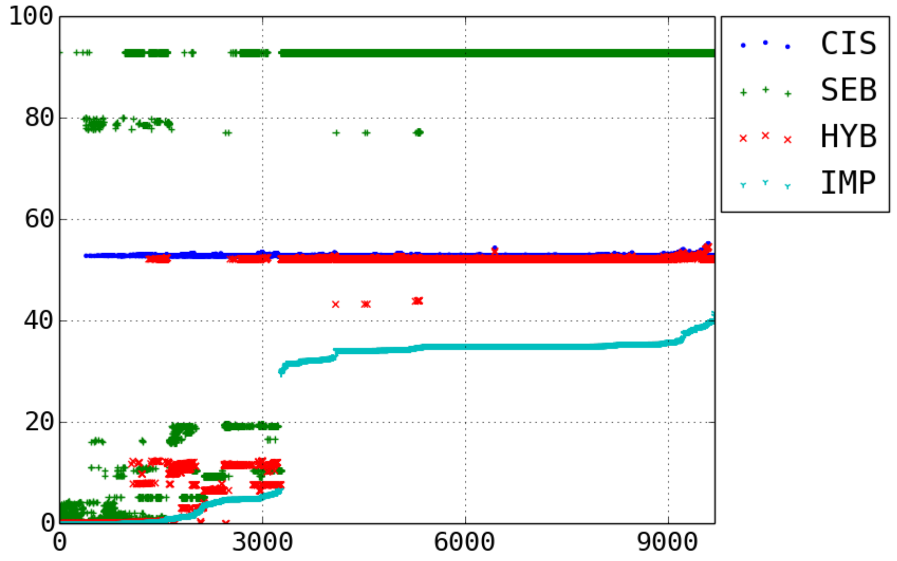
Figure 4.4: Time to Construct SDG in Configuration 1

ten times longer than that of CIS. The time required was still shorter than that required for the underlying analyses. After SDG construction, all four slicing techniques under comparison were able to compute a program slice in less than one second (a few milliseconds in most cases).

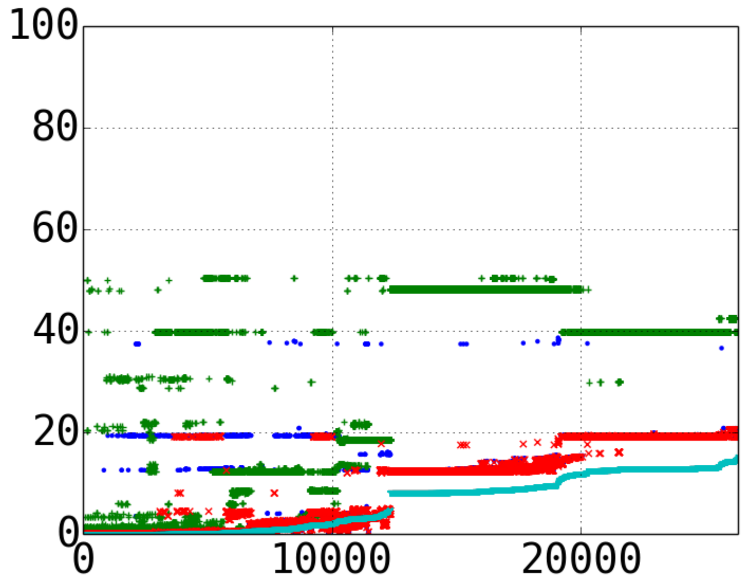
Table 4.5 shows the number of vertices and the number of edges for each SDG. IMP usually constructs a larger graph than CIS, to more precisely represent data-flow. CIS had 44 percent of the number of vertices that IMP had and 29 percent of the number of edges that IMP had.

Figures 4.5, 4.6 and 4.7 show the distribution of slice sizes of each application. The X-axis in these figures shows the index of slice criteria. It means that slices corresponding to the points in the same X-axis had the same criteria. The Y-axis shows the relative size of a program slice. The slices are sorted in ascending order of the relative IMP slice. The legend in the figures shows that CIS, SEB, HYB, and IMP are represented by blue, green, red, and light blue points, respectively.

IMP extracted smaller slices than the other three slicing techniques. For

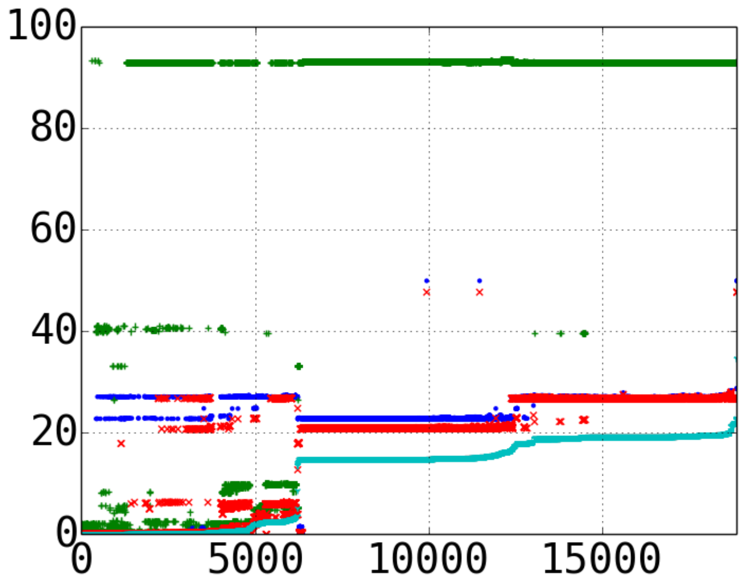


(a) avrora

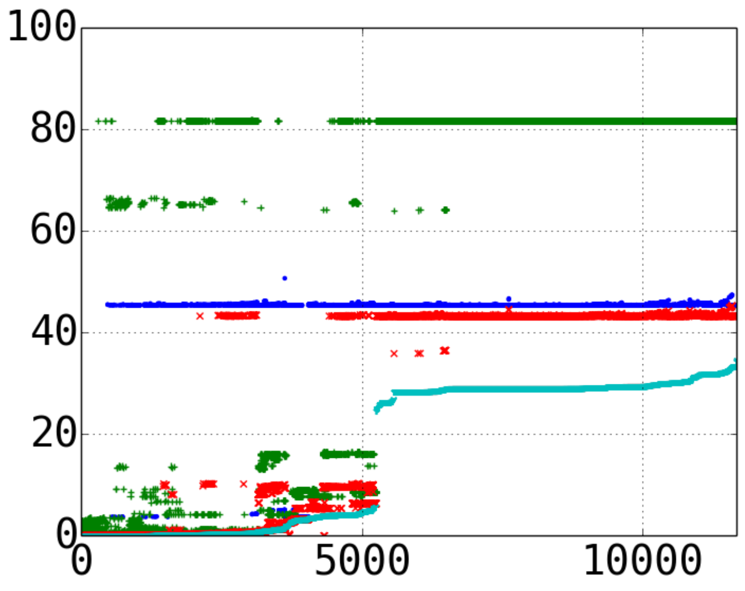


(b) batik

Figure 4.5: Scatter Plots of Relative Slice Size (percentage) on Configuration 1 (avrora and batik)

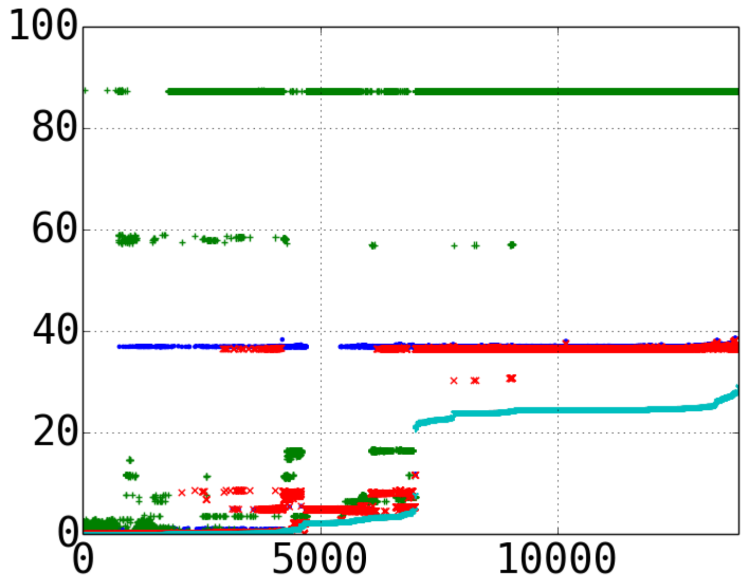


(a) h2

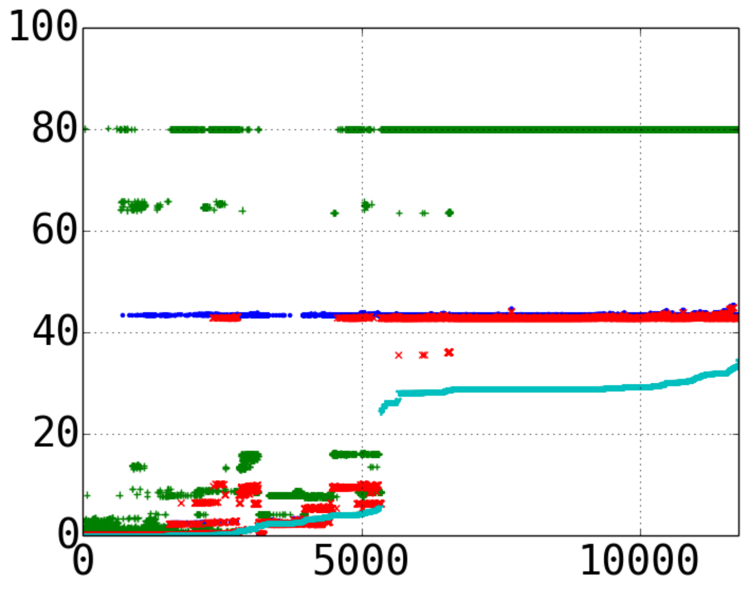


(b) luindex

Figure 4.6: Scatter Plots of Relative Slice Size (percentage) on Configuration 1 (h2 and luindex)



(a) pmd



(b) sunflow

Figure 4.7: Scatter Plots of Relative Slice Size (percentage) on Configuration 1 (pmd and sunflow)

Table 4.4: Detail of Time to Construct SDG in Configuration 1

| | Points-to Analysis (all) | Control-Flow Analysis (all) | Dependence Analysis (CIS/HYB) | Dependence Analysis (IMP) |
|---|---|---|---|---|
| avrora | 49.16 sec. | 0.18 sec. | 0.82 sec. | 7.84 sec. |
| batik | 193.49 sec. | 0.25 sec. | 2.75 sec. | 63.59 sec. |
| h2 | 59.76 sec. | 0.23 sec. | 1.97 sec. | 28.25 sec. |
| luindex | 51.82 sec. | 0.22 sec. | 0.91 sec. | 9.66 sec. |
| pmd | 62.81 sec. | 0.22 sec. | 1.07 sec. | 8.48 sec. |
| sunflow | 130.83 sec. | 0.13 sec. | 0.94 sec. | 12.40 sec. |

Table 4.5: SDG Size in Configuration 1

| | CIS | | IMP | |
|---|---|---|---|---|
| | #Vertices | #Edges | #Vertices | #Edges |
| avrora | 14,948 | 32,916 | 43,767 | 178,110 |
| batik | 38,439 | 86,394 | 84,052 | 276,873 |
| h2 | 30,291 | 64,538 | 151,738 | 1,054,257 |
| luindex | 18,388 | 39,701 | 47,957 | 181,496 |
| pmd | 22,782 | 48,096 | 50,322 | 175,634 |
| sunflow | 17,887 | 38,755 | 44,047 | 185,103 |

example, with avrora, IMP extracted less than 40 percent of the program instructions in 99 percent of the cases. IMP extracted 9.6 percent of the instructions on average. On the contrary, SEB extracted more than 80 percent of the instructions in all cases except for batik. One half of the instructions SEB extracted were likely false positives, according to IMP slice. An HYB slice was smaller than both SEB and CIS slices.

Figures 4.5, 4.6 and 4.7 show that several clusters of slices that had similar slice sizes are visible. A similar trend is reported in [90]. The phenomenon was caused by a dependence cluster [91], which is a strongly connected component in a graph. If traversal of program slicing reached an element in a dependence cluster, the traversal reached all elements in the dependence cluster.

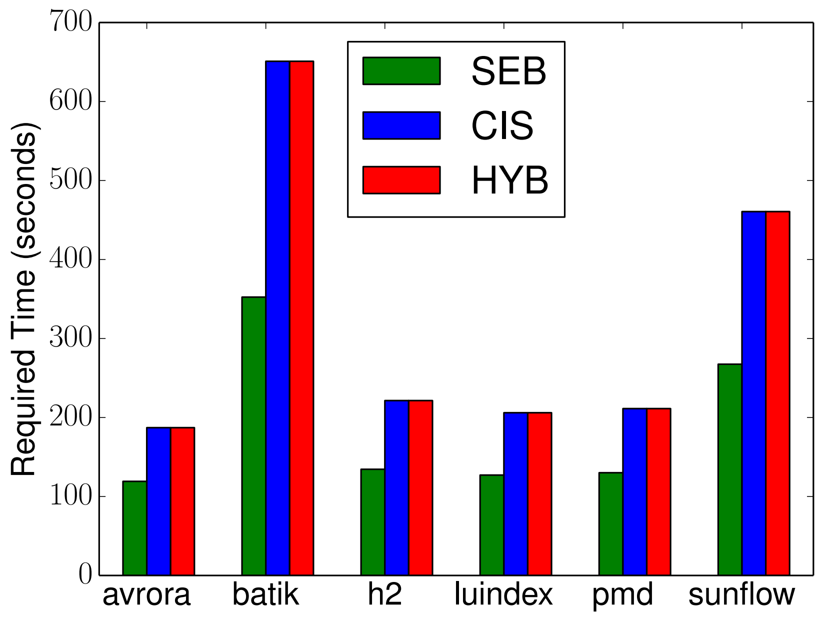
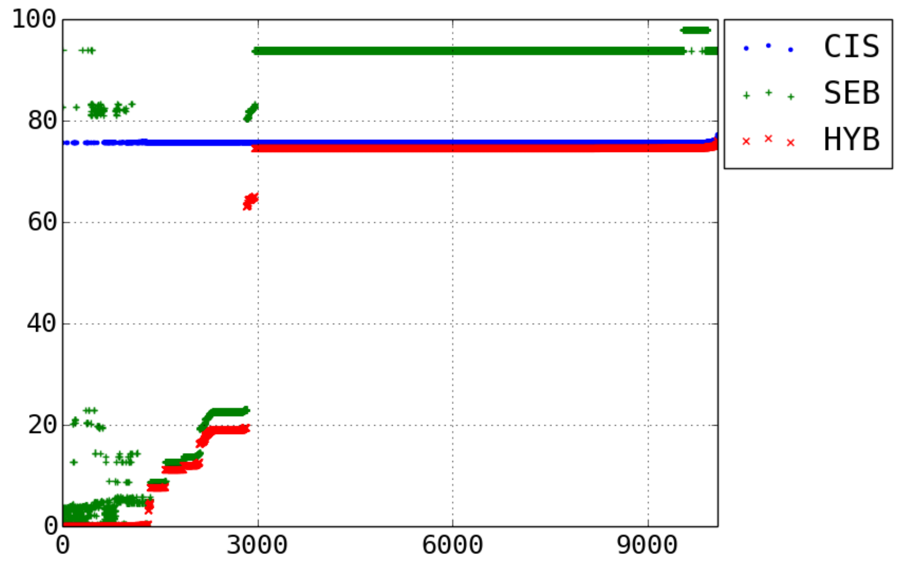


Figure 4.8: Time to Construct SDG in Configuration 2

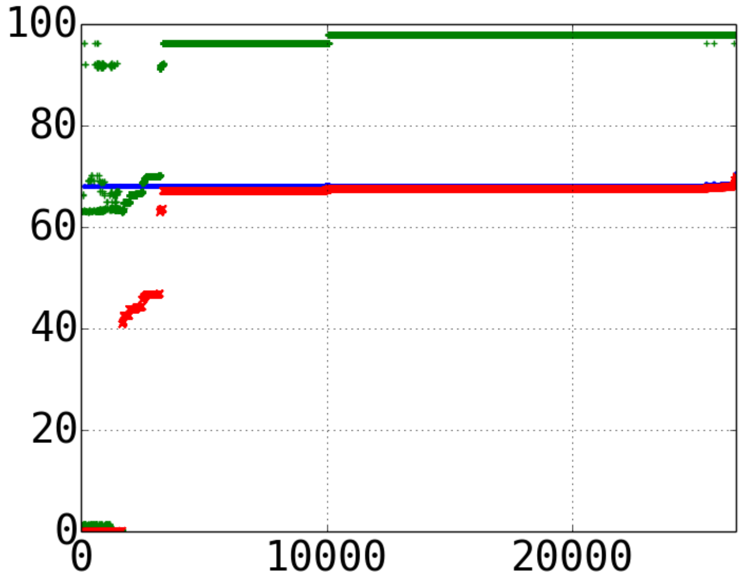
**Configuration 2: The Whole System Including Library**

Figure 4.8 shows the bar charts of the time required to construct an SDG for each application. Table 4.6 shows the time required to construct an SDG for an application including the library. The table does not include IMP, because 64GB of memory was insufficient for IMP to construct an SDG. Points-to analysis took several minutes, and SDG construction for CIS also took several minutes. SEB was much faster than CIS for a large-scale program, because the time for SDG construction increased significantly compared with points-to analysis. Table 4.7 shows the number of SDG vertices and edges for CIS. The numbers of vertices and edges were, on average, 28 and 40 times larger than SDG without the library, respectively.

The maximum time to compute a slice with SEB, CIS, and HYB was 4.27, 5.13, and 8.55 seconds, respectively. The results show that a program slice can be computed for a large program, within a practical time period. HYB computation took only a few seconds longer than CIS computation, because CIS had to construct a control-flow graph.

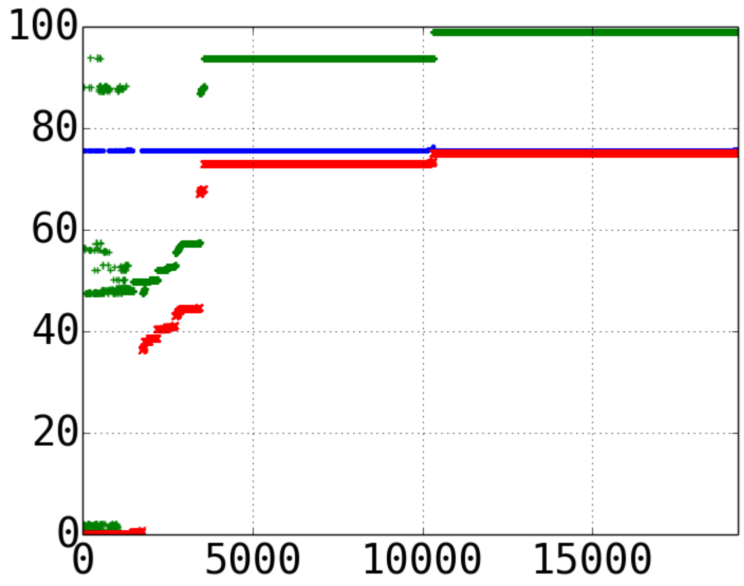


(a) avrora

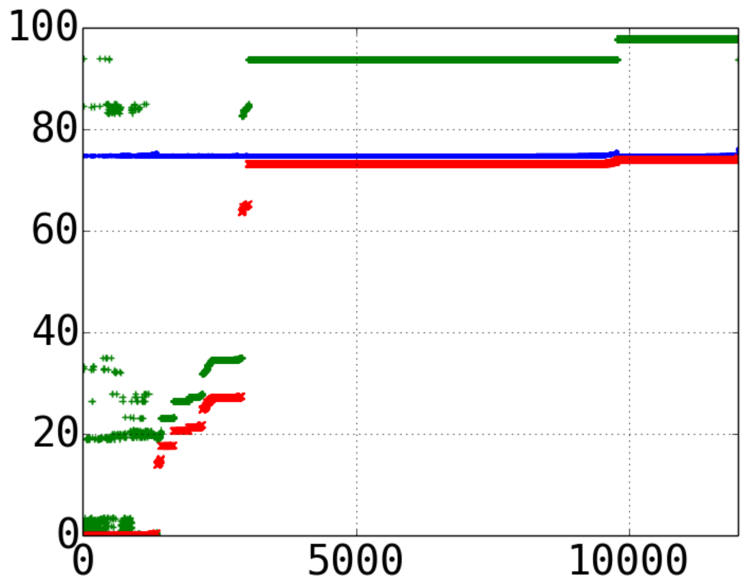


(b) batik

Figure 4.9: Scatter Plots of Relative Slice Size (percentage) on Configuration 2 (avrora and batik)

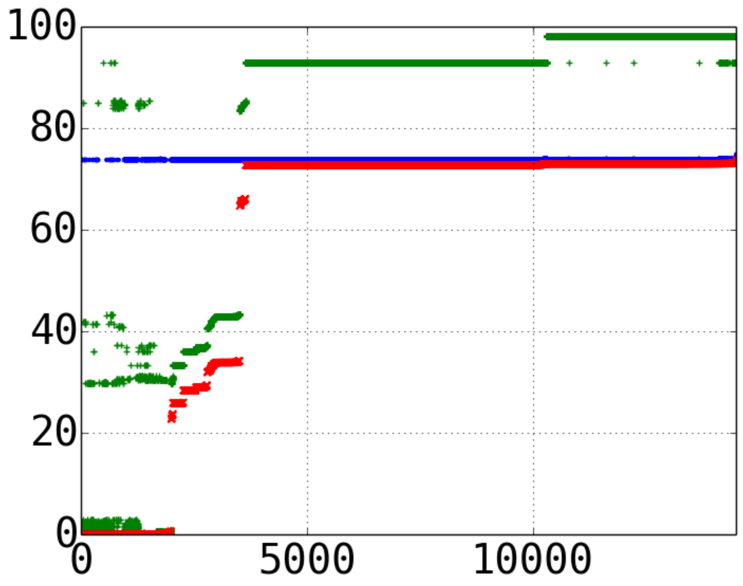


(a) h2

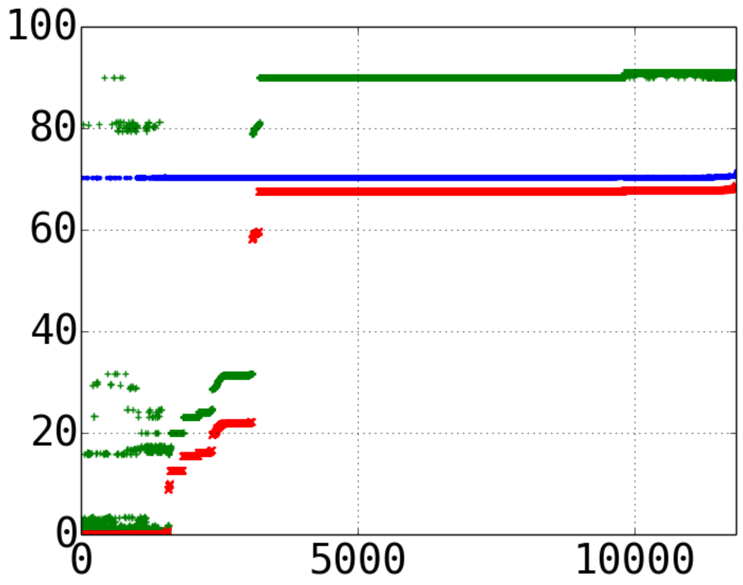


(b) luindex

Figure 4.10: Scatter Plots of Relative Slice Size (percentage) on Configuration 2 (h2 and luindex)



(a) pmd



(b) sunflow

Figure 4.11: Scatter Plots of Relative Slice Size (percentage) on Configuration 2 (pmd and sunflow)

Table 4.6: Detail of Time to Construct SDG in Configuration 2

| | Points-to Analysis (all) | Control-Flow Analysis (all) | Dependence Analysis (CIS/HYB) |
|---|---|---|---|
| avrora | 117.75 sec. | 1.40 sec. | 67.97 sec. |
| batik | 349.31 sec. | 3.10 sec. | 298.52 sec. |
| h2 | 132.21 sec. | 2.24 sec. | 86.96 sec. |
| luindex | 125.85 sec. | 1.24 sec. | 78.94 sec. |
| pmd | 128.67 sec. | 1.40 sec. | 81.19 sec. |
| sunflow | 264.57 sec. | 2.89 sec. | 193.19 sec. |

Table 4.7: Size of SDG for CIS in Configuration 2

| | #Vertices | #Edges |
|---|---|---|
| avrora | 449,186 | 1,335,862 |
| batik | 1,124,286 | 3,927,998 |
| h2 | 477,478 | 1,423,244 |
| luindex | 494,814 | 1,480,095 |
| pmd | 499,047 | 1,490,075 |
| sunflow | 1,033,777 | 3,207,852 |

Figures 4.9, 4.10 and 4.11 show the slice size distributions. The X-axis is the index of slices sorted by the ascending order of HYB slice size. The Y-axis shows the relative slice size. The relative slice size does not include the instructions in the library, because the library instructions are not visible to developers.

The resultant distributions are similar to the scatter plots of configuration 1. Although SEB extracted a small slice in some cases, it often extracted more than 80 percent of the instructions. Slices extracted by CIS were all almost the same size, due to SDG dependence clusters, while the maximum CIS slice size was lower than that of SEB. HYB extracted a significantly smaller slice in some cases, and a slightly smaller slice than CIS for most cases, because HYB was an intersection of SEB and CIS.

### 4.4.3 Scalability Analysis of Improved Slicer

#### Design and Analysis Targets

While IMP cannot analyze an entire system as mentioned previously, it is important to know the scalability for practical use. To measure the scalability of IMP, we measured required time to construct SDG for various configurations. If the construction takes longer than a predetermined threshold (3,600 seconds), the configuration is classified as *unanalyzable*. The threshold is determined by the length of developers' typical daily sessions [92].

We have used the same six applications as the first experiment. We selected 14 major sub-packages in LIB which are shown in Table 4.8. Table 4.8 shows the numbers of classes, methods, and instructions reachable from a `main` method of an application. Note that the numbers in Table 4.8 are taken from the result in the configuration: *the whole system including library.*

We constructed configurations for each application ($A$) with the set of sub-packages in Table 4.8 ($S$) by the following steps.

1. For each $s \in S$, one sub-package configuration including $A$ and $s$ is created.

2. Select sub-packages $S' \subseteq S$ whose one-package configurations are analyzable within the time limit.

3. For each $k \in \{2, 3, ..., |S'|\}$, a configuration is created by randomly selecting $k$ sub-packages from $S'$.

#### Result

Table 4.9 shows the distribution of analyzable programs and required time to prepare SDG. In order to show the difference of analyzability by the program size, we aggregate the result by the number of instructions. The interval is 10,000. The first column of Table 4.9 shows the range of the number of instructions. The column "#Config." shows the number of configurations whose size is included in the corresponding range. The column "#Analyzable (Percentage)" shows the number and ratio of the configurations which were analyzed within the limit time. The rightmost column shows the mean of required time to prepare SDG for the analyzable configuration programs. Note that all programs include over 100,000 instructions always failed to be analyzed, so that their results are totaled in the last row.

As Table 4.9 shown, even if a program is small size, the analysis time may be over 3,600 seconds. The minimum configuration which was not analyzed

Table 4.8: Sub packages used for scalability analysis

| Sub Package Name | #Classes | #Methods | #Instructions |
|---|---:|---:|---:|
| java.awt | 399 | 3,113 | 97,136 |
| java.beans | 20 | 192 | 5,792 |
| java.io | 92 | 805 | 21,089 |
| java.lang | 175 | 1,440 | 35,003 |
| java.math | 8 | 150 | 9,257 |
| java.net | 83 | 622 | 17,662 |
| java.nio | 127 | 623 | 10,156 |
| java.security | 108 | 482 | 11,400 |
| java.sql | 9 | 33 | 870 |
| java.text | 52 | 525 | 19,422 |
| java.util | 494 | 3,340 | 82,178 |
| javax.crypto | 27 | 166 | 5,365 |
| javax.security | 15 | 75 | 1,549 |
| javax.swing | 835 | 5,568 | 146,716 |

within 3,600 seconds is avrora with java.text package that has 26,325 instructions. Any configurations including java.text were not analyzed within the time limit, partly because java.text includes a recursive data structure such as a container for text that increases analysis cost.

The minimum unanalyzable configuration including more than one sub-packages is the configuration which includes h2, java.io, java.lang, java.math and java.security. The configuration includes 75,552 instructions. Since individual sub-packages are analyzable, the configuration is classified as unanalyzable because of the scalability rather than the complexity of one of the sub-packages. Actually, most of configurations including more than 80,00 instructions were classified as unanalyzable.

On the other hand, the maximum analyzable configuration is sunflow with java.awt, javax.crypto, and javax.security packages. The configuration includes 477 classes, 3,409 methods and 98,073 instructions.

Table 4.9 shows that programs including less than 70,000 instructions are analyzable in the highly probabilities which is at least 83.33%. On the contrary, the success probability of an analysis is slightly falling down if a program includes more than 70,000 instructions. The ratio of analyzable programs including from 70,000 to 80,000 instructions is about 55%. In addition, in the case of more than 80,000 instructions, the percentage of

Table 4.9: The Distribution of the Analyzable Programs and Time to Construct SDG

| Range of #Instructions | #Config. | #Analyzable (Percentage) | Mean of Time to Prepare SDG |
|---|---|---|---|
| [0,10k) | 0 | 0 (0%) | NA |
| [10k,20k) | 8 | 8 (100%) | 113.39 sec. |
| [20k,30k) | 28 | 25 (89.28%) | 121.35 sec. |
| [30k,40k) | 12 | 10 (83.33%) | 255.02 sec. |
| [40k,50k) | 11 | 10 (90.90%) | 416.16 sec. |
| [50k,60k) | 2 | 2 (100%) | 307.81 sec. |
| [60k,70k) | 3 | 3 (100%) | 837.48 sec. |
| [70k,80k) | 9 | 5 (55.55%) | 807.05 sec. |
| [80k,90k) | 9 | 1 (11.11%) | 1,934.65 sec. |
| [90k,100k) | 9 | 3 (33.33%) | 1,055.56 sec. |
| [100k,230k) | 14 | 0 (0%) | NA |

analyzable programs is more smaller. Therefore, 70,000 instructions seem to be the limit of analyzable program size by IMP of our implementation.

## 4.5 Discussion

We review the results in terms of accuracy and scalability to answer to RQ1.

First, we discuss accuracy. SEB in many cases extracted from 80 to 100 percent of the entire program. This indicates that SEB may have been only slightly effective. CIS consistently extracted 70 percent for any instructions in a program. This was caused by large SDG dependence clusters, as previously mentioned. HYB slices were sometimes much smaller than CIS slices, partly because SEB removed instructions in dependence clusters caused by infeasible SDG control-flow paths. IMP extracted smaller slices than the other techniques; however, it could not compute a slice for the entire program.

Table 4.10 shows the median relative slice sizes among the slicing techniques. IMP extracted 22 percent of the instructions that SEB extracted and 69 percent of the instructions that HYB extracted. Therefore, IMP would be the best choice of the four slicing techniques when using the library approximation.

Binkley et al. [40] report the average backward slice size as 28.1 percent

Table 4.10: Rates Compared Other Techniques

| | Configuration1 | Configuration 2 |
|---|---:|---:|
| HYB / SEB | 0.42 | 0.75 |
| HYB / CIS | 0.99 | 0.99 |
| IMP / SEB | 0.22 | - |
| IMP / HYB | 0.69 | - |

of the program, while IMP in this study had an average of nine percent. IMP might have achieved this improvement with more accurate field representations. In [40], they used another representation proposed by Liang et al. [35], the disadvantages of which are discussed in [24]. Differences between C/C++ and Java might also have affected the results. C/C++ programs can access arbitrary memory locations with pointers, while Java programs use only objects and fields to access memory locations. This difference could make Java program analysis easier than C/C++ analysis.

Second, we discuss scalability. IMP cannot analyze an entire system, as mentioned previously. IMP may construct a large tree of fields for an argument, because a field often contains another object. In the worst case, the size of a tree is estimated as $O(f^p)$, where $f$ is the number of fields in an object and $p$ is the number of object allocation sites. In other words, the number of vertices may increase exponentially according to program size. Actually, scalability anlaysis in Section 4.4.3 shows following two findings:

- If an analysis target program includes a container library, failure probability of the analysis will increase.
- 70,000 instructions is the indication of the analyzable program size of IMP. If a program includes more than 70,000 instructions, the success probability of analysis will be falling down. Moreover, 98,073 instructions is the limit size for analysis by IMP. The limit is large enough to analyze various programs in the DaCapo Benchmarks without libraries, while it is insufficient to analyze the programs with libraries.

On the contrary, since SEB needs only call graph and control-flow graph, the required memory space and analysis time are explicitly smaller than IMP. Similarly, the memory space of CIS/HYB depends on control/data dependencies of each method, call graph, and square of the field access instructions. The results showed that this cost is also practical, because CIS/HYB can be computed even if the analysis target includes Java Platform APIs.

We answer to RQ2 based on the previous discussion. Since IMP output the best precise result, IMP is suitable if an analysis target program can be analyzed. However, if the program size is larger than 70,00 instructions, the analysis is prone to fail. In addition, approximation of container libraries may be needed. Therefore, it is suitable that developers need to analyze a middle-size application or a subsystem of a large application. Note that development of a suitable approximation for an application may present technical challenges.

On the other hand, when developers need to analyze a large program or a program including a library, e.g. Java Platform API, HYB is the most suitable technique, in terms of scalability and accuracy. CIS analyzes control and data dependencies in HYB, and SEB takes a few additional seconds to remove infeasible control-flow paths from CIS.

## 4.6 Threats to Validity

We approximated Java library code in the comparison of IMP and the other techniques, due to IMP's scalability. If a method in a Java library provided dependence via heap fields, IMP and CIS slices will not include statements that actually relate to the selected statement. The approximation assumes methods in the library do not call back application methods. This assumption may also result in false negatives. We did not directly compare the results of these two configurations, due to this threat.

Points-to analysis affects slices. We used Spark, but there are many other points-to analysis tools and methods we could have used: e.g. object-sensitive points-to analysis [81], hybrid context-sensitive points-to analysis [82], etc. The resulting slices could be more precise if more accurate points-to analysis is used. Improving points-to analysis would be more effective for CIS and IMP, rather than SEB, because SEB uses only a call graph created by points-to analysis, while CIS and IMP use improved field access information, in addition to the call graph.

Reflection handling also affects analysis results. We used TamiFlex, which collects methods that were actually invoked during execution of a target program. The results are affected not only by the analysis target programs, but also by their execution. Program slices computed in the study miss some statements that could be executed through reflection, but are excluded from the default execution scenario in the DaCapo Benchmarks.

We obtained the results from six applications. The results may not be applicable to arbitrary Java programs; however the target programs included real Java applications. Therefore, we believe the result is indicative of a general trend.

The results of this experiment depend on the experimental environment which includes CPU, RAM, and operating system. Therefore, the concrete values such as seconds of execution may not be applicable to general execute environments. However, the result of relatively comparing each technique is generally applicable, such as the ratio of difference of the precision and the analysis time.

Our slice implementations might contain some defects possibly affecting the results shown here. We have provided our implementation and dataset on our website, to enable other researchers to replicate the study and conduct further research.

## 4.7 Related Work

Binkly et al. [40] compare various kinds of program slicing. However, IMP and SEB were not included in the comparison. Additionally, the target applications are written in C/C++. Our targets were programs written in Java. In addition, CodeSurfer which is a program slicing tool and used in [40] can analyze C++ templates in analysis target source code, but it does not analyze templates in library and not in analysis target source code, such Standard Template Library (STL). [40] does not clear about handling STL. On the contrary, our Java bytecode analysis includes Java Platform APIs corresponding to STL can be performed.

[22] also compares program slicing and SEB and analyzes programs written in C/C++. We have compared SEB and two slicing techniques, CIS and IMP, for Java programs. Beszedes et al. [23] compared static execute after analysis with forward program slicing for C/C++ and Java programs. However, they did not evaluate SEB and the backward slicing techniques that we have compared in this study.

Binkley et.al. also evaluated optimization techniques of *massive slicing* which is an application of program slicing [93]. Massive slicing takes all program slice from each possible criteria, and is used for debugging. Optimization techniques evaluated in [93] optimize SDG or memory representation for SDG, but it is required that building full SDG for the first time. We evaluate approaches of building SDG or one corresponding SDG before optimizing.

[24] evaluates the SDG size of programs. However, most of the analyzed programs are student programs, and the analyzed program size is at most 10

KLOC, while we analyzed real Java applications. They also did not report the average size of a slice compared with other techniques.

There are slicing approaches for object-oriented programming languages other than IMP. Liang et al. [35] propose an SDG that also simulates object trees of formal/actual in/out vertices. However, they expand field trees based on the object type (i.e. variable type) and use k-limit to expand a tree to stop infinite expansion. Therefore, the approach is less precise than IMP. Larsen et al. [36] also proposes an SDG that treats heap field input/output as formal/actual in/out. This means that the Larsen approach does not make an object tree, it is field-based, not object-sensitive. IMP is an object-sensitive approach; therefore, it is more precise than Larsen's approach.

## 4.8 Summary

We evaluated the scalability, precision and tradeoffs of four slicing techniques; SEB, CIS, HYB, and IMP. We selected six real Java applications in DaCapo benchmarks for evaluation. We computed slices of instructions in each application and compared the slice sizes and computation times of the slicing techniques.

The results show that HYB had good scalability, which was achieved with a small cost increase over CIS. An HYB slice is 25 percent smaller than the SEB slice. Moreover, an HYB slice is sometimes significantly smaller than a CIS slice, because HYB considers feasible control-flow paths from SEB. When developers need to analyze a large program or a program including a library, our results indicate that HYB is suitable.

On the other hand, our results show that IMP is the most accurate of the four slicing techniques. An IMP slice contains 22 percent of SEB and 69 percent of HYB. However, IMP does not have the scalability required to analyze a large program, such as a whole system including a JDK library. When developers need to analyze a middle-size program or a subsystem, our results indicate that IMP is suitable.

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

In this dissertation, we have addressed three issues related to software reuse, which include software license, retrieval for reusable components implementing a feature, extraction of a reusable component.

First, we have performed a quantitative study for the investigation of the impact of software licenses to copy-and-paste reuse activity among OSS projects. The study has shown the followings:

- copy-and-paste mostly occurs within source code under the same license
- substantial amount of reused code fragments appeared in *GPLv2+* product
- permissive licensed files tend to be reused more than copyleft ones

The result of the first study gives the quantitative guide for developers' license selection.

Secondly, we proposed an approach for building domain specific verb-object relationship dictionaries. The dictionary includes tuples consisting of (Verb, Object, Indirect Object) which are extracted from identifiers in method signatures. In the experiment, we showed that the tuples in the dictionary are popular in the target domain or common Java programs. The result of the second study helps appropriate naming and resolving searching issue.

Thirdly, we performed a comparative study of four backward program slicing techniques for Java. The comparison shows that the combination of context-insensitive slicing and static execute before has the scalability for analyzing an entire large system, and outputs a slice which size is 25 percent smaller than static execute before, on average. On the other hand, improved slicing has the scalability for analyzing middle size applications, and is smaller than 31 percent compared to the combination technique. The results of the third study help selection of the program slicing which is appropriate to a situation for a developer who wants to extract a component from an existing system.

We believe that the results of these studies improve reuse activities from both sides of a reusable product developer and a user of a reusable component.

## 5.2 Future Work

Some future work is needed for support of distributing reusable product from the points of view of license and identifier names. Future work of the license study is making license selection support tool which provides the impact and future of popularity of a product per license. Regarding identifier names, current researches provide precise identifier names by natural language processing. However, these studies targeted English identifier names, despite many programming languages enable to use Unicode characters for identifier names. Therefore, future work includes support of non-English names such as Japanese.

On the other hand, future work for support of extraction process includes proposing more lightweight slicing techniques for daily uses. Another direction includes the improvement of representation of a program slice for confirming that an extracted component is appropriate for a reuse purpose, such as a representation by natural language like summary comments for a method [94].

# Bibliography

[1] Commons collections. `http://commons.apache.org/proper/commons-collections/` (Accessed December 2014).

[2] SourceForge.net. `http://sourceforge.net/` (Accessed December 2014).

[3] Github. `https://github.com/` (Accessed December 2014).

[4] Open Source Initiative. `http://www.opensource.org/` (Accessed December 2014).

[5] Open Source Initiative. Open Source Definition. `http://www.opensource.org/docs/osd` (Accessed December 2014).

[6] Open Source Initiative. The BSD license. `http://opensource.org/licenses/BSD-3-Clause` (Accessed December 2014).

[7] Apache Software Foundation. Apache License, Version 2.0. `http://www.apache.org/licenses/LICENSE-2.0` (Accessed December 2014).

[8] Free Software Foundation. GNU general public license. `http://www.gnu.org/licenses/gpl.html` (Accessed December 2014).

[9] Epson pulls linux software following gpl violations. `http://beta.slashdot.org/story/02/09/11/2225212/epson-pulls-linux-software-following-gpl-violations` (Accessed December 2014).

[10] Playstation 2 game ico violates the gpl. `http://news.slashdot.org/story/07/11/28/0328215/playstation-2-game-ico-violates-the-gpl` (Accessed December 2014).

[11] GitHub Inc. 10 million repositories. https://github.com/blog/1724-10-million-repositories (Accessed December 2014).

[12] Ryuji Shimada, Makoto Ichii, Yasuhiro Hayase, Makoto Matsushita, and Katsuro Inoue. A-score: Software component recommendation system based on source code under development (in japanese). 50(12):3095–3107, December 2009.

[13] Tetsuo Yamamoto, Norihiro Yoshida, and Yoshiki Higo. Seamless code reuse with source code corpus. In *20th Asia-Pacific Software Engineering Conference*, volume 2, pages 31–36, 2013.

[14] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transaction on Software Engineering*, 22(6):424–437, 1996.

[15] Nancy Pennington. Empirical studies of programmers: second workshop. pages 100–113. Ablex Publishing Corp., 1987.

[16] Victor Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajerski, Gerald Page, and Sharon Waligora. The software engineering laboratory: An operational software experience factory. In *Proceedings of the 14th International Conference on Software Engineering*, pages 370–381, 1992.

[17] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[18] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[19] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph based program slicing. *Software Engineering, IEEE Transactions on*, 23(4):246–259, Apr 1997.

[20] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 33–42, 2003.

[21] Andreas Marx, Fabian Beck, and Stephan Diehl. Computer-aided extraction of software components. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 183–192, October 2010.

[22] J. Jász, A. Beszedes, T. Gyimothy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the International Conference on Software Maintenance*, pages 137–146, Sept 2008.

[23] A. Beszedes, T. Gergely, J. Jasz, G. Toth, T. Gyimothy, and V. Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 295–304, Oct 2007.

[24] Christian Hammer and Gregor Snelting. An improved slicer for java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 17–22, 2004.

[25] Debian Project. Debian GNU/Linux. `http://www.debian.org/` (Accessed December 2014).

[26] Daniel M. German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 437–446, 2010.

[27] Toshihiro Kamiya. CCFinder Official Site. `http://www.ccfinder.net/ccfinderx.html` (Accessed December 2014).

[28] Yasuhiro Hayase, Makoto Ichii, and Katsuro Inoue. A novel approach for building a thesaurus for program comprehension (in japanese). In *Proceedings of winter workshop 2008 in Dogo*, number 3, pages 33–34, 2008.

[29] Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, pages 9–18, 1996.

[30] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, March 2002.

[31] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[32] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17(8):751–761, Aug 1991.

[33] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[34] Neil Walkinshaw, Marc Roper, Murray Wood, and Neil Walkinshaw Marc Roper. The java system dependence graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64, 2003.

[35] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367, 1998.

[36] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, 1996.

[37] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 44–54, 2003.

[38] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–18, 2013.

[39] iTR co. Inc. Press release. `http://www.itr.co.jp/company_outline/press_release/130919PR/index.html` (Accessed December 2014).

[40] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2), April 2007.

[41] Grammatech's codesurfer. `http://www.grammatech.com/research/technologies/codesurfer` Accessed December 2014.

[42] Walt Scacchi. Free/open source software development: Recent research results and emerging opportunities. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 459–468, 2007.

[43] Free Software Foundation. What is copyleft? `http://www.gnu.org/copyleft/copyleft.en.html` (Accessed December 2014).

[44] Michel Ruffin and Christof Ebert. Using Open Source Software in Product Development: A Primer. *IEEE Software*, 21(1):82–86, 2004.

[45] Yu Kashima, Yasuhiro Hayase, Norihiro Yoshida, Yuki Manabe, and Katsuro Inoue. A preliminary study on impact of software licenses on copy-and-paste reuse. In *Proceedings of the International Workshop on Empirical Software Engineering in Practice*, pages 47–52, 2010.

[46] Open Source Initiative. Open source licenses. `http://www.opensource.org/licenses` (Accessed December 2014).

[47] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with Off-the-Shelf Components: 10 Facts. *IEEE Software*, 26:80–87, 2009.

[48] Free Software Foundation. Various licenses and comments about them. `http://www.gnu.org/licenses/license-list.en.html` (Accessed December 2014).

[49] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transaction on Software Engineering*, 32(3):176–192, March 2006.

[50] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[51] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[52] Daniel M. German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90, 2009.

[53] Hung-Fu Chang and Audris Mockus. Evaluation of source code copy detection methods on freebsd. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 61–66, 2008.

[54] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Method and implementation for investigating code clones in a software system. *Information & Software Technology*, 49(9-10):985–998, 2007.

[55] Makoto Ichii, Takashi Ishio, and Katsuro Inoue. Cross-application fan-in analysis for finding application-specific concerns. In *Proceedings of the Fourth Asian Workshop on Aspect-Oriented Software Development*, pages 39–43, 2008.

[56] J.S. Poulin. The search for a general reusability metric. In *Proceedings of the Workshop on Reuse and the NASA Software Strategic Plan*, 1996.

[57] Judith Barnard. A new reusability metric for object-oriented software. *Software Quality Control*, 7:35–50, May 1998.

[58] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys*, 28:415–435, June 1996.

[59] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering*, pages 106–115, 2007.

[60] YongLee Yii, Yasuhiro Hayase, Makoto Matsushita, and Katsuro Inoue. Token Comparison Approach to Detect Code Clone-related Bugs. *Technical report of IEICE. SS*, 107(505):37–42, 2008.

[61] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: report to our respondents. In *Proceedings of GUIDE 48*, April 1983.

[62] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[63] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[64] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[65] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 3–12, 2006.

[66] Oracle Corporation. The java tutorial. `http://java.sun.com/docs/books/tutorial/java/javaOO/index.html` (Accessed December 2014).

[67] Microsoft. Method naming guidelines. `http://msdn.microsoft.com/en-us/library/4df752aw(v=vs.71).aspx` (Accessed December 2014).

[68] Yuya Onizuka, Yasuhiro Hayase, Takashi Ishio, and Katsuro Inoue. Supporting method naming for java programs using verb-object relations (in japanese). In *IEICE Technical Report*, volume 111, pages 1–6, 2012.

[69] Sun Microsysytems. Java platform, standard edition 6 api specification. `http://java.sun.com/javase/6/docs/api` (Accessed December 2014).

[70] Zachary Fry, David Shepherd, Emily Hill, Lori Pollock, and K Vijay-Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *IET Software*, 2(1):27–36, 2008.

[71] The Apache Software Foudation. Opennlp. `http://opennlp.apache.org/` (Accessed December 2014).

[72] David Shepherd, Lori Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 3–14, 2006.

[73] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242, 2009.

[74] Einar W. Høst and Bjarte M. Ostvold. The programmer's lexicon, volume i: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2007.

[75] Einar W. Høst and Bjarte M. Ostvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 294–317, 2009.

[76] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, October 2009.

[77] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 746–755, 2011.

[78] Sebastian Danicic, Chris Fox, Mark Harman, Rob Hierons, John Howroyd, and Michael R. Laurence. Static program slicing algorithms are minimal for free liberal program schemas. *The Computer Journal*, 48(6):737–748, November 2005.

[79] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[80] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Department of Computer Science, University of Copenhagen, May 1994.

[81] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.

[82] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Con-*

*ference on Programming Language Design and Implementation*, pages 423–434, 2013.

[83] 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 Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[84] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20, 1994.

[85] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, 2003.

[86] 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*, pages 13–. IBM Press, 1999.

[87] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[88] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, 2011.

[89] Eric Bodden. DacapoAndSoot. `https://code.google.com/p/tamiflex/wiki/DaCapoAndSoot` (Accessed December 2014).

[90] Judit Jász, Lajos Schrettner, Arpád Beszedes, Csaba Osztrogonác, and Tibor Gyimothy. Impact analysis using static execute after in webkit. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, pages 95–104, 2012.

[91] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 177–186, 2005.

[92] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Control*, 19(1):5–34, March 2011.

[93] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 30(1), November 2007.

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