2013

Code Clone Discovery Based on Concolic Analysis

Daniel Edward Krutz
Nova Southeastern University, dan7800@yahoo.com

Code Clone Discovery Based on Concolic Analysis

by

Daniel Edward Krutz

A dissertation report submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University
2012

Code clone Discovery Based on Concolic Analysis

by
Daniel Edward Krutz
Fall 2012

Software is often large, complicated and expensive to build and maintain. Redundant code can make these applications even more costly and difficult to maintain. Duplicated code is often introduced into these systems for a variety of reasons. Some of which include developer churn, deficient developer application comprehension and lack of adherence to proper development practices.

Code redundancy has several adverse effects on a software application including an increased size of the codebase and inconsistent developer changes due to elevated program comprehension needs. A *code clone* is defined as multiple code fragments that produce similar results when given the same input. There are generally four types of clones that are recognized. They range from simple type-1 and 2 clones, to the more complicated type-3 and 4 clones. Numerous clone detection mechanisms are able to identify the simpler types of code clone candidates, but far fewer claim the ability to find the more difficult type-3 clones. Before CCCD, MeCC and FCD were the only clone detection techniques capable of finding type-4 clones. A drawback of MeCC is the excessive time required to detect clones and the likely exploration of an unreasonably large number of possible paths. FCD requires extensive amounts of random data and a significant period of time in order to discover clones.

This dissertation presents a new process for discovering code clones known as Concolic Code Clone Discovery (CCCD). This technique discovers code clone candidates based on the functionality of the application, not its syntactical nature. This means that things like naming conventions and comments in the source code have no effect on the proposed clone detection process. CCCD finds clones by first performing concolic analysis on the targeted source code. Concolic analysis combines concrete and symbolic execution in order to traverse all possible paths of the targeted program. These paths are represented by the generated concolic output. A *diff* tool is then used to determine if the concolic output for a method is identical to the output produced for another method. Duplicated output is indicative of a code clone.

CCCD was validated against several open source applications along with clones of all four types as defined by previous research. The results demonstrate that CCCD was able to detect all types of clone candidates with a high level of accuracy.

In the future, CCCD will be used to examine how software developers work with type-3 and type-4 clones. CCCD will also be applied to various areas of security research, including intrusion detection mechanisms.

Acknowledgments

First and foremost I would like to thank my family. Without their guidance and support, I would not be the man I am today. I would like to especially thank my grandparents. Lydia, Edward (Opal), Betty, Karl, Louise (Lou-Lou) and Bob [1]. Lydia, thanks for all the card games (I still say you were cheating somehow) and all the fantastic ravioli dinners. "Opal", thanks for all the hotdogs and hot chocolates at halftime of the football games. Betty, thanks for the swimming and day trips to the Adirondacks, baseball games and always keeping me more than well fed. Karl, thanks for all the wonderful stories and always being so friendly, calm, hardworking and inspirational. Louise, thanks for keeping my dressed so well and all the pancakes for breakfast. Bob, thanks for always being one of my best friends, so easy to talk to and the countless Matchbox cars.

Over the years, I have come to realize how lucky I've been.

I would also like to thank my advisor, Dr. Frank Mitropoulos. He taught the first class at Nova which really appealed to me and helped to foster my love of Software Engineering. My committee has also been wonderful. I would especially like to thank Dr. Ronald M. Krawitz. Over the years, he has been a wonderful mentor, classmate, committee member, but most of all friend. There were countless phone calls that he gave me advice or simply words of encouragement to carry on.

I would also like to thank the wonderful study group at Nova. They were extremely helpful in helping me to finish this degree. I've made many good lifelong friends from this group.

[1]"The Bonda"

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Background

Software systems may be large, complicated and used for extended periods of time. During the lifecycle of an application, it will typically need to be constantly updated and maintained. This may need to be done in order to fix bugs, keep it compatible with new technologies or improve performance (Kim, Jung, Kim, & Yi, 2011; Singh & Goel, 2007; Ueda, Kamiya, Kusumoto, & Inoue, 2002).

Source code is often reused throughout the application. As the application's source code is maintained and evolves, additional reuse occurs (Marcus & Maletic, 2001; Singh, Bhatia, & Sangwan, 2009). Additionally, people will typically join and leave the software development team throughout this lifecycle. This is known as *developer churn*. This developer churn means that developers with varying levels of understanding of the application will be modifying the application (Meneely, Williams, Snipes, & Osborne, 2008; Monden, Nakae, Kamiya, Sato, & Matsumoto, 2002). These developers will likely have to alter the application in numerous locations during this maintenance phase. Some of these changes and added functionality will be redundant across the application. From a software engineering standpoint, the most appropriate way to make these alterations is to refactor the application, and not perform simple copy and pastes of the code (Pressman, 2010).

Developers are generally aware of the adverse effects that copying and pasting code snippets throughout the application will have. However, it still occurs quite frequently in most large software applications. Between 5 -23% of all code in software is estimated to be redundant or exact copy and pastes of source code (Baxter, Yahin, Moura, Sant'Anna, & Bier, 1998; Schulze, Apel, & Kastner, 2010). Code redundancies are created for both necessary and unnecessary reasons. On some occasions, redundancies may be necessary due to language restrictions. In other scenarios, they may occur simply due a lack of system understanding or other avoidable situations (Ducasse, Rieger, & Demeyer, 1999; Jarzabek & Xue, 2010).

A *code clone* is defined as multiple code fragments which produce similar results when given the same input (Fukushima et al., 2009). There are four types of code clones. These range in complexity from type-1 to type-4 clones. Type-1 clones are the simplest, and most easy to detect, while type-3 and 4 clones are much more complicated, and much harder to detect, if at all discoverable (Roy, Cordy, & Koschke, 2009). Specific examples of each type are described later in this work.

Code clones are undesirable in software for several reasons. First of all, when making changes on the cloned segments, these alterations will need to be made uniformly throughout the application. Failure to do so may lead to faults being created inside of the application or even logical errors when some of the clones are repaired and others continue to contain bugs. The maintenance costs of the application will also likely be increased since alterations made to redundant segments will have to be done numerous times (Juergens, Deissenboeck, Hummel, & Wagner, 2009). In economic terms, this increased maintenance cost is a very serious matter. It is estimated that maintenance activity comprises over 80% of the overall cost of many software projects (Shukla & Misra, 2008). Finally, tangled or scattered clones across a system will make it very difficult for developers to understand how specific functions are implemented throughout a system. In the event the developer does not understand the implementation of a specific module of code, they may unintentionally change the functionality of the system during software maintenance (Maisikeli & Mitropoulos, 2010).

Problem Statement

Throughout the software development cycle, code cloning is a frequent occurrence and is generally considered to be a sign of bad software design (Duala-Ekoko & Robillard, 2010; Wettel & Marinescu, 2005). Most often, clones are the result of a copy and paste activity by the developers. This action is one where the same code segment is replicated throughout the application for various reasons (Deissenboeck, Hummel, & Juergens, 2010). With the progression of time, applications are generally growing larger. As the number of lines in the source code continues to expand, detecting clones becomes more difficult (Bruntink, Deursen, Engelen, & Tourwe, 2005).

The issue of the existence of clones in applications is not a new problem (Baker, 1995). Clones themselves do not introduce faults into the system. Faults are introduced because an application will generally need to be maintained. It is during this maintenance phase is where clones generally have the largest adverse effect on the system. The maintenance phase of an application generally represents 40-70% of the total cost of the project (Ducasse et al., 1999; Seaman, 2008; Ueda et al., 2002).

Clones significantly add to the expense of the software maintenance phase of a project (Juergens et al., 2009). Inconsistently changing clones in the application is where a significant increase to the overall maintenance cost of the application may occur (Hummel, Juergens, & Steidl, 2011). Clones also increase the general size of the application (Deissenboeck et al., 2010). This makes locating desired sections of code much more difficult. This can be a significant issue since locating these specific sections for bug fixing can be a difficult and time consuming task (Chen, Jaygarl, Yang, & Wu, 2008).

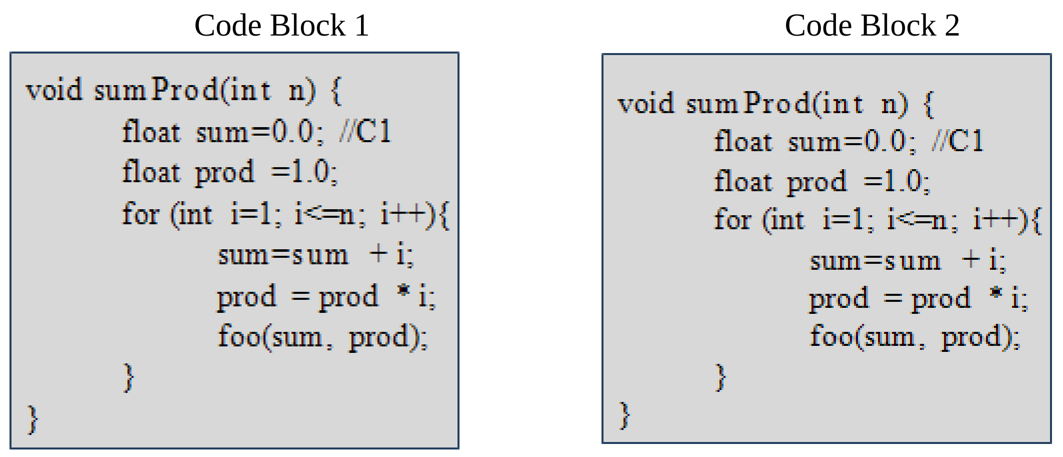
When numerous clones exist, developers need to pay extra care in changing all clones uniformly (Krawitz, 2012). Inconsistent changes to clones represent faults in 50% of the cases if the change was introduced intentionally (Deissenboeck et al., 2010). The existence of clones may also lead to significant segments of dead code, or unused segments in the application. This may be lead to problems with comprehensiveness, readability, extensibility, and maintainability (Kapser & Godfrey, 2008).

An additional goal of software development is to create applications which are highly modular. Some of the benefits of code modularization are reusability, and ease of maintenance. The software testing process may also be hindered by the presence of code clones. If unit testing is utilized, extra unit tests will be required to be written against each of these code clones. This will add extra time to the project initially, as well as to the maintenance of these tests. Error discovery and location may also be hindered by the existence of code clones (Roy et al., 2009).

Code clones represent a significant problem for applications, and it is important to be able to identify these clones so they may be dealt with accordingly by the developers (Kapser & Godfrey, 2008). There are four defined levels of code clones as described by Gold, Krinke, Harman, and Binkley (2010):

- Type-1: The code is syntactically identical except for white spaces, layout and comments.
- Type-2: Code is syntactically identical except for variations in identifiers, literals, types, and variations permitted under Type 1.
- Type-3: Code which is modified by adding, removing, or alteration statements, in addition to variations allowed under Type 2.
- Type-4: Code which uses different syntax, but produces the same result.

As described by Roy (2009), Figure 1 represents a type-1 code clone. The two sections of code are identical in every manner. Figure 2 represents a type-2 clone. Only the variable identifiers have been altered and are shown in bold. Figure 3 represents a type-3 code clone. The only difference is the extra input variable into the *foo* method. The rest of the syntax and functionality remains identical to the original source code. Figure 4 and Figure 5 represent type-4 clones. The declaration order of *prod* and *sum* have been reversed. However, the remaining code has identical syntax to the original.



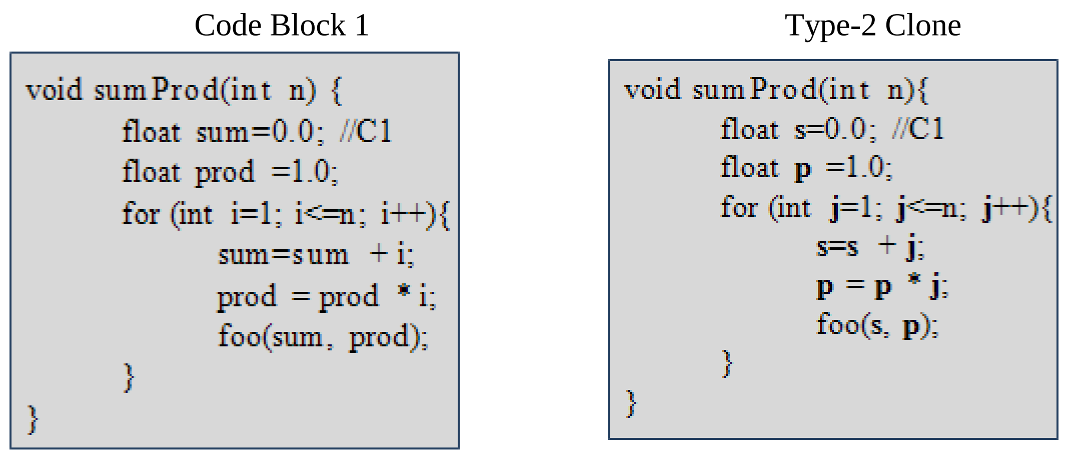
Code Block 1

```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

Code Block 2

```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

Figure 1. Type-1 clone example (Roy et al., 2009).



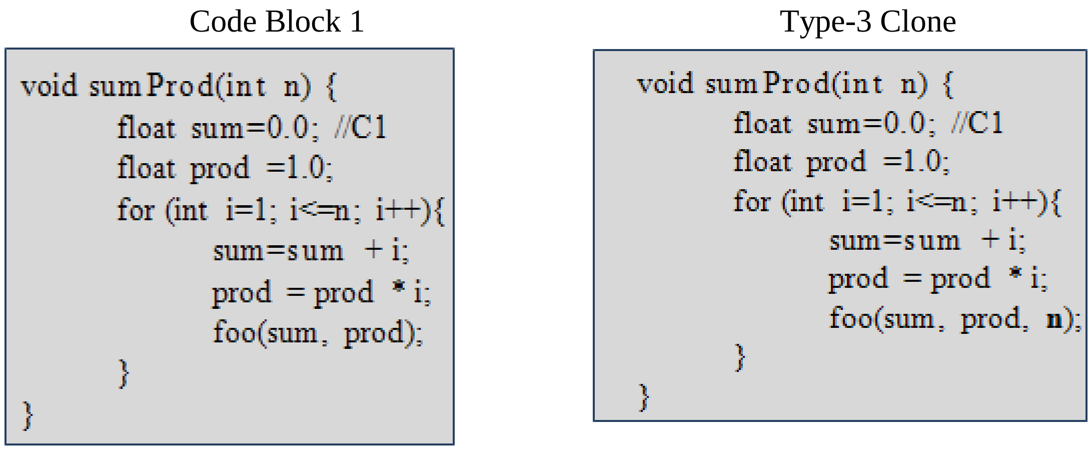
Code Block 1

```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

Type-2 Clone

```
void sumProd(int n){
        float s=0.0; //C1
        float p =1.0;
        for (int j=1; j<=n; j++){
                s=s + j;
                p = p * j;
                foo(s, p);
        }
}
```

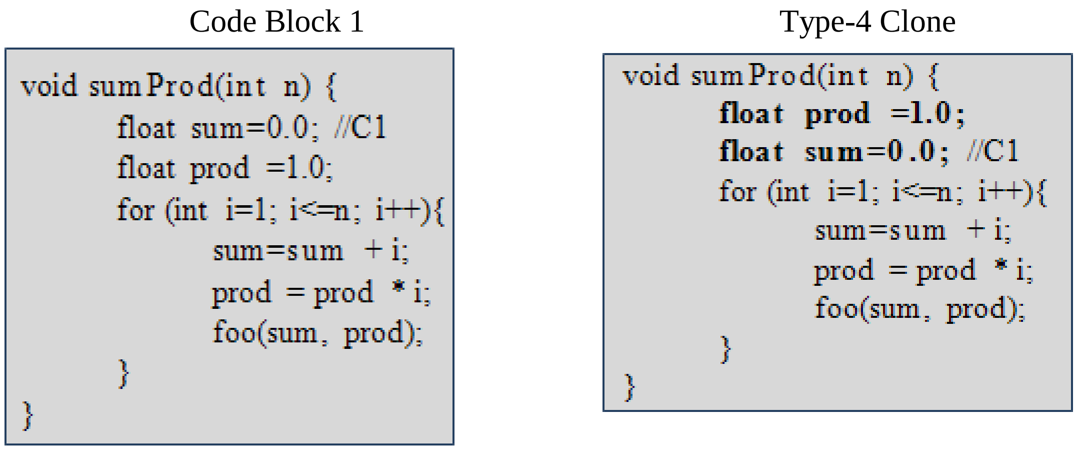
Figure 2. Type-2 Clone example (Roy et al., 2009).



```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod, n);
        }
}
```

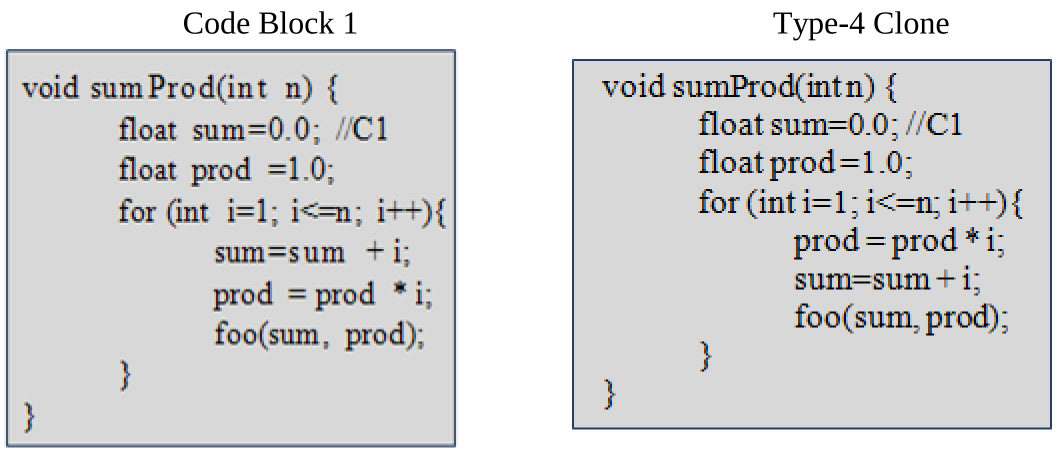
Figure 3. Type-3 Code Clone (Roy et al., 2009).



```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

```
void sumProd(int n) {
        float prod =1.0;
        float sum=0.0; //C1
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

Figure 4. Type-4 Code Clone (Roy et al., 2009).



```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod =1.0;
        for (int i=1; i<=n; i++){
                sum=sum + i;
                prod = prod * i;
                foo(sum, prod);
        }
}
```

```
void sumProd(int n) {
        float sum=0.0; //C1
        float prod=1.0;
        for (int i=1; i<=n; i++){
                prod = prod * i;
                sum=sum + i;
                foo(sum, prod);
        }
}
```

Figure 5. Type-4 Code clone (Roy et al., 2009).

In order to detect various clone levels, there are currently numerous methods which have been implemented by both the research and commercial communities (Deissenboeck et al., 2010; Higo, Kamiya, Kusumoto, & Inoue, 2007). Most contemporary research concentrates on discovering Type-1 and 2 clones, and is largely successful at doing so. A few of these tools are able to detect Type-3 clones (Roy et al., 2009). Added, modified or deleted statements often alter the functionality of a software component. Redundant code of this nature often causes type-3 and type-4 clones. These are typically difficult for clone detection techniques to detect and many methodologies do not even attempt to find them (Koschke, 2007).

In symbolic analysis, program variables that typically contain concrete values are replaced with symbolic values. These are inputs which may represent a wide range of possible values. Traditionally, symbolic execution has been used to explore a large number of possible program paths (Pasareanu et al., 2008; Person, Dwyer, Elbaum, & Pasareanu, 2008; Person, Yang, Rungta, & Khurshid, 2011). Concolic execution uses both concrete and symbolic values for interpreting a target program. Concrete states allow concolic analysis to deterministically evaluate any program expression. This helps to overcome some limitations of pure symbolic analysis such as the inability to handle some types of loops, recursion and exploration of infeasible paths (Takaki et al., 2010). Applications which contain code clones are generally poorly designed and are more expensive in terms of maintenance, extensibility and ease of comprehension (Roy et al., 2009).

While numerous clone detection methods exist, they all suffer for a variety of reasons. One of the most prevalent issues is the inability for most techniques to efficiently and effectively detect type-3 and type-4 clones. Only two methods claim the ability to detect type-4 clones. One of which utilizes functional analysis, while the other uses a memory comparison technique. A primary drawback of functional analysis is that random data needs to be generated in order to discover code clones. This can be a difficult and time consuming process (Krawitz, 2012). An issue with the memory comparison based process is that it takes quite a long time to run since it uses a standard static analysis technique. Other problems with standard static analysis include the exploration of an unreasonably large number of program states and the substantial cost of maintaining and solving symbolic constraints along the program’s execution paths (Kim et al., 2011; Majumdar & Sen, 2007). Additional problems also exist with other existing methodologies. These include normalization and need for historical data (Basit & Jarzabek, 2005). Due to these issues with current approaches, further work is required in order to create a robust technique for code clone procedure.

Clone detection is important in aiding the software development process in a variety of ways. While numerous techniques are able to detect type-1 and type-2 clones, few are able to discover type-3. Even less claim the ability to detect type-4 clone candidates. A new and robust technique for clone discovery is needed to fill this gap.

Dissertation Goal

The goal of this dissertation was to discover clone candidates using concolic analysis. No existing clone detection techniques appear to utilize this method. The proposed technique discovers code clone candidates based on their functional nature. Naming conventions, comments and other syntactical attributes have no bearing on the clone candidate detection process. This is accomplished by analyzing the concolic output of the source code. This dissertation proposed a process known as Concolic Code Clone Discovery (CCCD) in order to discover clones.

Research Questions

The primary research question CCCD answered was if concolic analysis could be used to detect code clones. No previous work appears to have ever attempted this. A secondary research question was what types of clones concolic analysis would be able to discover.

Relevance and Significance

Today’s large software systems are complicated applications which have the capability of being heavily utilized in industry for extended periods of time. During the lifespan of an application, it is very likely to require extensive modifications. In order lifespan of an application, it is very likely to require extensive modifications. In order for its users to remain satisfied, the software will need to be constantly evolving (Geiger, Fluri, Gall, & Pinzger, 2006). Specific reasons for these updates include altering the program’s functionality, bug fixes, and environment changes. Based on this, the possible negative ramifications of code clones are very important. The effort required to perform changes on a system go up as do the number of code clones. This means that code clones are a significant factor which must be paid attention as the system evolves (Geiger et al., 2006).

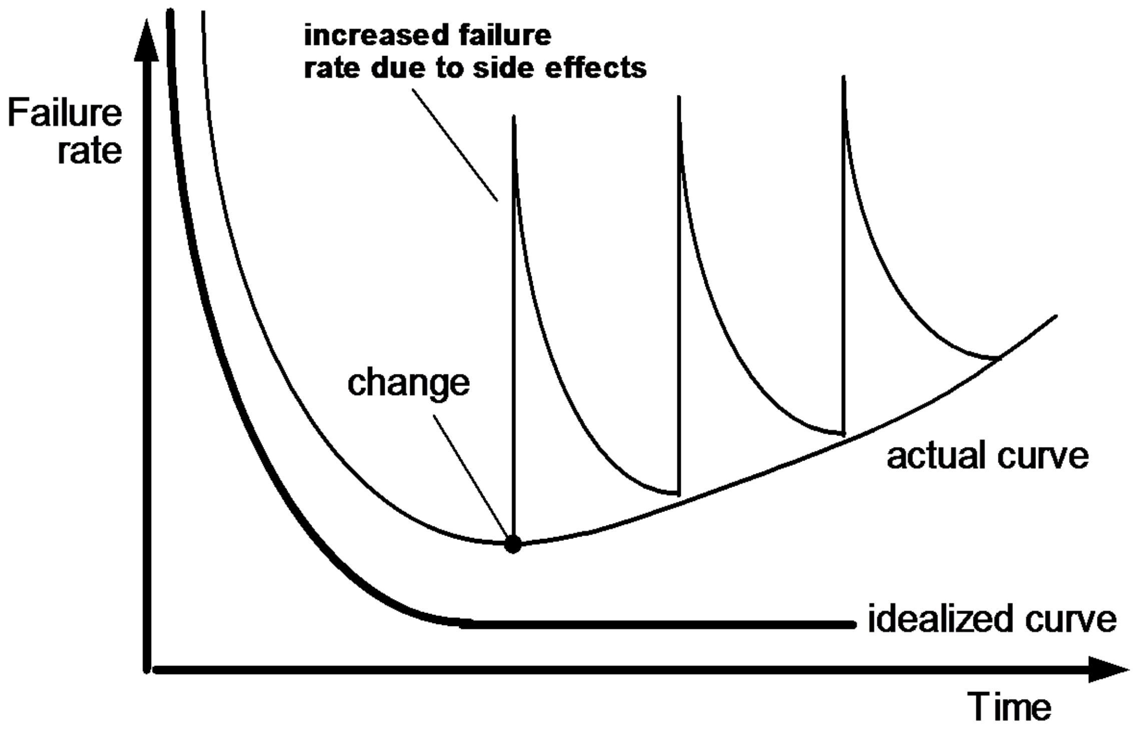


Figure 6. Failure curves for software (Pressman, 2010).

Dealing with large software systems is extremely challenging for the companies who must maintain them. As maintenance is performed on the system, it generally becomes harder to maintain in a quality manner (Figure 6). Ideally, the failure rate of software should go down, or at the very least remain steady. However, this is not the case. As maintenance is performed on the application, the error rate actually tends to rise (Monden et al., 2002; Pressman, 2010) .

Larger applications will likely have more developers associated with them, and thus a higher turnover rate. As developers join and exit the project, they will not only develop in their individual manner, but will also not be aware of existing functionality in other parts of the application (Monden et al., 2002). This means that these developers will have a high likelihood of knowingly or unknowingly injecting a high level of clones into the application. Once this happens, it will be very difficult to retain a high level of maintainability and reliability for the system (Akito & Shinichi, 2001). Very often, code clones may be introduced for valid reasons. Typically however, they most often exist as a result of a poor software design or poor development practices (Fukushima et al., 2009).

Since the existence of code clones tends to help contribute to this high cost of change, locating them can be extremely beneficial in decreasing the already significant maintenance portion of an application (Geiger et al., 2006). There are several existing techniques for detecting code clones (Bruntink et al., 2005; Kim et al., 2011; Krawitz, 2012; Roy et al., 2009):

- Text: Attempt to detect similar sequences by using minimal analysis.
- Lexical/Token: Apply lexical analysis to the source code and attempts to locate similar lines of code.
- Tree: Obtain a syntactical representation of the source code by using parsers.
- Metrics: Related to hashing algorithms. In this methodology, each fragment of a program, a number of various metrics are gathered regarding them. This information is subsequently used to find similar fragments.
- Graph: Obtains source code representation from a high level of abstraction. Program Dependency Graphs (PDGs) are comprised of information of a

semantic manner. These include data such as control and data flow of the program.

- Functional: Performs black box testing on blocks of code. Clones produce identical outputs when provided identical inputs.

- Symbolic: Uses symbolic output of an application to discover similarities.

Typically, text, token and tree based methodologies focus on the source code as it is being developed. Graph based techniques rely upon the data and control flow information for clone discovery. Finally, metrics based methods use a hybrid of various existing techniques and gauge the results by using vectors, graphs or other abstract representation (Bruntink et al., 2005; Roy et al., 2009).

CCCD is a new technique for clone candidate discovery. This identification process is done without paying attention to the comments or naming conventions in the source code. It is done entirely through the concolic analysis of the application. Additionally, concolic analysis was demonstrated to be a new and practical solution to candidate code clone discovery. This is something which does not appear to have been previously attempted. Type-4 clones are comprised of different source code. CCCD is capable of discovering these types of clones. This is because concolic analysis only follows the flow and functionality of the program and is not directly tied into the syntax of the source code.

Based on a literature review, no attempts have been made to discover code clones candidates (of any type) using concolic analysis. CCCD is well suited to discover these clones because current techniques rely upon text based comparisons, source code analysis, data flow analysis or symbolic analysis. Since CCCD uses concolic analysis which combines concrete and symbolic values, it only cares about the flow of the application. This means that numerous problematic issues which have hindered previous clone detection techniques are irrelevant to the proposed technique. Problematic areas such as comments and naming conventions which have plagued many existing clone detection techniques have no effect on CCCD. Knowing the flow of the application is important because these paths help define the functional equivalence of two code segments. Two segments which are functionally equivalent are clones (Person et al., 2008). CCCD is beneficial because it limits the negative ramifications of code clones on applications, and thus reduces the development, and maintenance costs while helping to assure a high quality application.

The goal of this dissertation was to develop a process to locate code clones by comparing the output of the concolic analysis of various code segments. This helps to address the problems of the high cost of software maintenance and poor overall software quality. The cost of maintaining the software will decrease as the number of code clones is abated. CCCD allows software developers the ability to find clone candidates.

Barriers and Issues

CCCD is a new and complicated approach to clone discovery. There were numerous challenges which had to be overcome. These included hurdles related to the gathering of test data, the evaluation of existing tools, concolic analysis aspects, and any minor concolic equalization processes which needed to be carried out.

Clones have the potential to be a wide range of sizes. Some clones may only be a few lines long (Bruntink, Deursen, & Tourwe, 2004), or more than 200 lines long (Monden et al., 2002). This means that CCCD needed to account for these widely varying sizes. Large software projects are not generally developed by a single developer. Due to this, an innumerable amount of different development techniques and processes had to be accounted for by the proposed discovery process.

In order to perform concolic analysis on a system, it must first have an existing concolic analysis tool be able to be adapted to run each class individually using concolic analysis. CCCD had to be made to append the name of the instantiating class onto the blocks of generated symbolic data. These are all important for the comparison process and will be discussed in the approach section. Additional modifications had to be performed on the selected concolic analysis tool. Some of these were fairly substantial and required a significant amount of development effort. Other alterations were much simpler changes to configuration settings, but required a significant amount of thought and background work in order to ensure that they were configured properly.

Once the necessary concolic values have been generated, a *diff* was performed on this output. A *diff* is a simple operation that notes any differences between two files. Any discovered similarities are an indication of a possible clone candidate. This comparison was done using an already existing tool, Notepad++. Future work may combine all of this functionality into a single application, but at the present time it is out of the scope of this dissertation.

The output is compared at the method level, meaning that clones are only detectable at this level of granularity. This is because each method in the application has concolic analysis performed upon it individually by a modified version of Java Path Finder (JPF), an application which was originally created by NASA. Presently, this altered tool is only capable of accurately and effectively generating the data required for CCCD on a method by method basis. This is the same for all currently known concolic analysis applications. This may be a limitation of CCCD since clones can exist within multiple methods, when the methods themselves are not clones. Further enhancements to existing concolic analysis tools would eliminate the need to only analyze code on a method by method basis. It is important to note that this is a limitation of the tools, not of the overall concolic analysis technique.

Limitations

The purpose of this dissertation was to identify code clone candidates using concolic analysis. CCCD discovered code clone candidates in a manner which is independent of the semantics of the code being analyzed. Even though CCCD represents a new and robust method for discovering clones, it does have a potential limitation. While concolic analysis is able to overcome many of the path constraints of symbolic analysis, other restrictions exist which may limit its ability to perform complete analysis on a system. One of these inhibitors is the inability of current concolic analysis tools to compute concrete values to satisfy all constraints (Sen, 2007). This did not pose a problem for the conducted research, but could create problems if CCCD were attempted to be implemented on a much larger scale. This is an inherent problem with existing concolic analysis tools which may be fixed by future research into this specific area. However, this is out of the scope of this dissertation.

Definition of Terms

Abstract Syntax Tree-Based Techniques	Use parsers to obtain a syntactical representation of the source code, usually in the form as an AST, before the AST is searched for similar sub-trees (Krawitz, 2012, p.15).
AST	Abstract Syntax Tree (Krawitz, 2012, p.15).
Code Clone	Implementing the same program functionality more than one time. Multiple code fragments that produce similar results given the same input (Krawitz, 2012, p.15).
Concolic Analysis	Combines random testing and symbolic execution to partly remove the limitations of random testing and symbolic execution based testing (Sen, 2007, p. 1).
Functional Behavior	How the output of a system is affected by inputs without regard for the contents of the system. Ignoring the internal mechanism of a system and focusing on the outputs generated in response to inputs. Black Box behavior (Krawitz, 2012, p.15).
Lexical	Relating to words or the vocabulary of the system as distinguished from the syntax rules and construction (Krawitz, 2012, p.16).
Lexical/Token-Based Techniques	Applies lexical analysis to the source code and use the lexical analysis to find similar lines of code (Krawitz, 2012, p.16).

PDG	Program Dependence Graph (Krawitz, 2012, p.16).
Program Dependence Graph-Based Techniques	Obtain a source code representation of high abstraction that contain information such as control and data flow of the program that can be analyzed and compared to find code clones (Krawitz, 2012, p.16).
Program Maintenance	The modification of a system after delivery to correct faults or improve performance. Most maintenance implements functional enhancements (Krawitz, 2012, p.15).
Random Testing	Random testing generates a large number of inputs randomly. The program is then run on those inputs to check if programmer written assertions hold, or in the absence of specifications (Majumdar & Sen, 2007, p. 1).
Refactor	Changing the source code of a computer program without modifying the program's functional behavior (Krawitz, 2012, p.17).
Semantically Similar	Two blocks of code that have the same meaning or produce the same results based on an analysis of the words and symbols used to generate the source code (Krawitz, 2012, p.17).
Source Code	A collection of human-readable statements that provide instructions to the computer so it can complete a task. Also called a program (Krawitz, 2012, p.17).
Symbolic Execution	A program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an execution path (Sen, 2005, p. 1).
Syntactically Similar	The same results based on an analysis of code metrics or AST analysis (Krawitz, 2012, p.17).
Text-Based Techniques	Perform minimal analysis before attempting to detect similar sequences of lines of code (Krawitz, 2012, p.17).

Summary

Duplicated source code in an application is known as code clones. These can have several adverse effects on an application. Some of which include increased maintenance and code comprehension costs. Most research recognizes four types of code clones. Type-1 and Type-2 clones are reasonably basic and detectable by the majority of clone detection mechanisms. Only a few works claim the ability to detect the more elaborate Type-3 clones while even less state that they are able to identify Type-4 clones, which are the most complex.

CCCD is a new system for detecting code clone candidates. Concolic analysis was used in order to discover similar functionality within an application. Things that have plagued many previous clone detection systems, such as semantics, were not taken into consideration and therefore do not pose a problem for the proposed process. CCCD begins by analyzing the target application and producing the necessary concolic values. This output is then examined for identical sections, which is indicative of a clone candidate. CCCD ultimately provided an indication of the clone candidate along with the locations of all candidates in the examined source code.

Chapter 2

Review of the Literature

Testing is heavily used in industry in order ensure software quality. Having the ability to automatically traverse all paths of an application is important for numerous testing techniques (Sen, 2007). Manually testing all paths of an application is generally not practical due to the sheer number of possibilities that even the smallest applications may have. Exhaustive analysis or the testing of all possible paths cannot be reasonably expected to be feasible, even in an automated fashion for the vast majority of applications. In order to adequately test applications, several types of analysis techniques that aim to examine all possible paths of the application have been devised (Majumdar & Sen, 2007).

Code Clone Detection

The majority of current clone detection techniques do an adequate job of finding type-1, type-2, and sometimes type-3 clones (Koschke, 2007). Only two works claim the ability the ability to detect type-4 clones (Kim et al., 2011; Krawitz, 2012). In order to properly understand CCCD, it is important to understand the previous methodologies along with their strengths and weaknesses. Additionally, it is important to understand the effectiveness of the existing techniques in order to be able to compare them with CCCD. Previous research by Roy and Cordy (2008) separate the existing detection techniques into four principle categories. These are text, lexical, syntactical and Semantic.

Textual Approach

Text based approaches use very little or no normalization or transformation on the source code being examined. White space and comments are usually ignored. Typically, the raw source code is directly used in the clone detection process (Bruntink et al., 2005; Roy et al., 2009). One approach examines the substrings of the source code. The first step is to hash the code fragments of a defined number of lines. A sliding window technique is then used to identify sequences of lines that have the same hash values as clones (Johnson, 1993). Dot or scatter plot approaches have also been utilized with arguably better results (Roy et al., 2009). In this process, a coordinate value is assigned to various source code segments. If two lines have the same coordinate plots, they are assumed to be equal. This process has the additional benefit of allowing clone information to be visualized. (Ducasse et al., 1999; Roy et al., 2009). A drawback to text based approaches is that by examining the source code directly, small changes may have significantly adverse effects on this system since it is essentially using a pattern matching scheme in order to discover clones (Bruntink et al., 2005).

Lexical Approach

Lexical approaches first transform the source code into a series of lexical "tokens." This is done by using a compiler-style lexical analysis technique (Roy & Cordy, 2008). The list of generated tokens is then scanned for duplications. Duplications are then considered to represent code clones. A primary benefit of this method over text based approaches is that minor changes such as formatting, spacing and code renaming generally pose a smaller problem. Variations of this technique have been found to detect type-1, type-2 and in certain situations type-3 clones (Roy et al., 2009). A powerful tool which uses this approach is CCFinder (Kamiya, Kusumoto, & Inoue, 2002).

Syntactical Approach

Syntactical approaches first use a parser in order to convert programs into either parse trees or Abstract Syntax Trees (ASTs) (Roy et al., 2009). Tree based approaches discover clones by using a parser to examine the generated AST (Bruntink et al., 2005). Similar sub trees are then discovered using tree-matching techniques. The discovered code segments are then returned as classes or clone pairs. A more sophisticated clone detection process may be done by abstracting variable names, literal values, and other tokens in the tree representation. However, there are some issues with this technique. Possible problematic areas include near misses between functions, sub clones and errors caused by scale (Baxter et al., 1998; Roy & Cordy, 2008). Several AST detection techniques have been proposed thus far. Various methods include dynamic programming approaches for handing differences in comparing sub trees (W. Yang, 1991). Converting the AST to XML using a data mining technique in order to extract parameterized clones has also been proven to be beneficial (Wahler, Seipel, Gudenberg, & Fischer, 2004).

Metrics based approaches gather various metrics for each of the code fragments. Instead of comparing the code directly, these metric vectors are examined (Lague, Proulx, Mayrand, Merlo, & Hudepohl, 1997; Mayrand, Leblanc, & Merlo, 1996; Roy & Cordy, 2008). Euclidean distance and other distance evaluators may be utilized in order to indicate code similarities (Koschke, 2007). One technique utilizes calculated metrics for syntactical units. These may include class, function or a method which generate values which may be compared to discover clones in these units (Roy & Cordy, 2008).

Semantic Approach

Semantic approaches utilize static program analysis to generate more precise information than from simply using syntactic similarities. This technique is broken up into two categories, Program Dependency Graph (PDG) and hybrid approaches (Roy & Cordy, 2008). PDG Based Techniques consider semantic information encoded in the dependency graph as a form of source code abstraction. In this technique, the generated information in the dependency graph represents control and data flow information (Bruntink et al., 2005). A sub graph isomorphic algorithm is used to discover clones as similar sub graphs from the PDG (Roy & Cordy, 2008).

Symbolic-based Approach

A recent approach to code clone discovery has been through the use of a process known as Memory Comparison-based Clone Detection (MeCC). This technique compares abstract memory states which are generated by a semantic-based static analyzer. In order to generate all of the necessary memory states, symbolic analysis is used to estimate the effects on all of the procedures being examined.

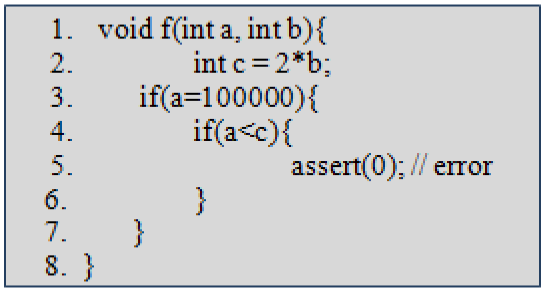
Behavior-based Approach

Behavior based approaches attempt to discover code clone candidates by studying the functional behavior of a block of code. This is done by examining how blocks of code react to various inputs. Inputs were provided to methods in the source code and the output of these methods are then recorded. Similar output indicates a clone candidate (Krawitz, 2012). Krawitz (2012) created a functional analysis tool that discovers clones using a processes known as Code Clone Discovery Based on Functional Behavior. This work claims to be able to detect all types of clones and is completely independent of the syntax of the source code being analyzed.

Concolic Analysis

Concrete variables are items which have a specific value assigned to them. Symbolic analysis involves symbolic variables used in place of concrete values for input. These symbolic values may represent theoretically any possible value in the system. A primary goal of symbolic analysis is to discover all feasible system paths (Sen, Marinov, & Agha, 2005). Concolic analysis combines concrete and symbolic values in order to traverse all possible paths of an application (up to a given length). The main premise behind symbolic execution is the use of symbolic values instead of actual concrete values (Sen, 2007). Symbolic analysis has been used to compare two programs for semantic equality (Menon, Pingali, & Mateev, 2003). The computed symbolic outputs are expressed as a function of the symbolic inputs (Cadar et al., 2011). The state of a symbolically executed program is comprised of several values. These include the path condition (PC), the program counter, and the symbolic values of the program variables. According to Pasareanu (2008), it is comprised of a Boolean formula over the symbolic inputs. The program counter states the next statement which is to be executed. The various paths followed during a program's symbolic execution is represented by the *symbolic execution tree* (Khurshid, Pasareanu, & Visser, 2003). The next statement to be executed is typically defined by the program position (PP) (Person et al., 2008).

Concolic analysis is a variant of symbolic execution where concrete executions are run simultaneously with symbolic analysis (Majumdar & Sen, 2007). The concolic execution process begins by first generating random values for primitive inputs, and null values for pointer inputs. Using a loop, these values are fed into the targeted method. Following this execution, a new test input is generated using the symbolic constraint in the path constraint. Using this information, solvers are generated which are used to generate new test input in order to direct the application along a different execution path. This process is continued until all possible distinct paths have been reached using a depth search strategy (Sen, 2007). The primary advantage of using concolic instead of symbolic analysis techniques is the presence of concrete values. These can be used to simplify constraints and help in the precise reasoning of complex data structures (Majumdar & Sen, 2007).



```
void f(int a, int b){
        int c = 2*b;
    if(a=100000){
        if(a<c){
                assert(0); // error
        }
    }
}
```

Figure 7. Code to be examined by concolic analysis.

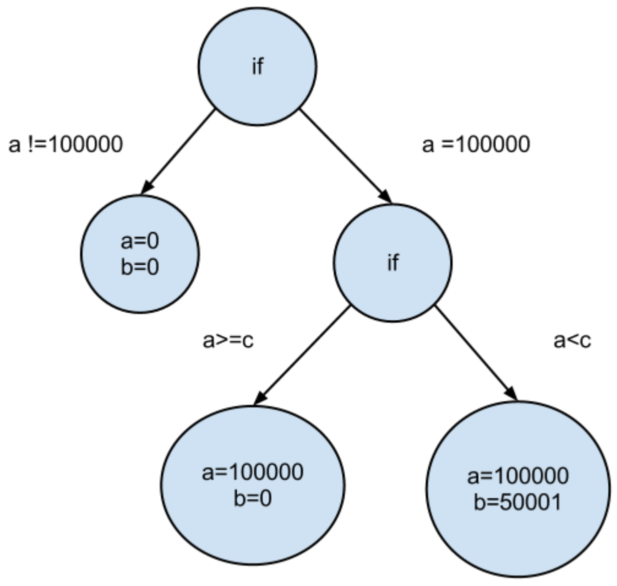
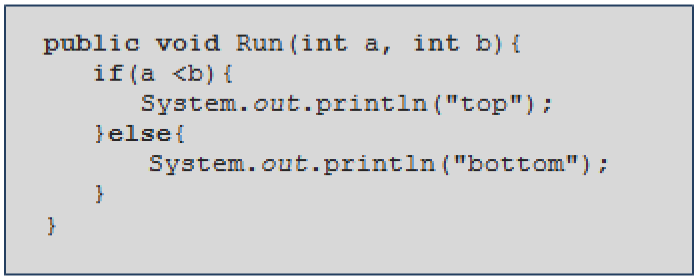


Figure 8. Concolic Analysis Flow

Figure 7 displays a function which is to have concolic analysis performed upon it, while Figure 8 shows its data flow. The analysis process would first begin with an arbitrary value being assigned to a and b. For the concrete execution, a=b=1. Line #2 would set c to be 2, and the *if* statement in the 3rd line will fail since a ≠ 100000. The symbolic execution will follow the same path taken by the concrete execution, but will merely treat a and b as symbolic variables. C will be set to the expression 2b and will make note that a ≠ 100000 since the test in line 3 failed. This is known as a *path condition* and will need to be true for every execution following this same path.

The goal is to follow every path of the application. This means that the next step for this example is to take the last path condition encountered, a ≠ 100000 and negate it. This means that a=100000. In order to find values for the input variables a and b, an automated theorem prover is then invoked using a complete set of symbolic and path variables created during the symbolic execution process. This automated theorem prover shows the logical consequence of a set of statements. The goal of this prover is to help ensure that all program paths are properly followed. In this situation, the values created by this theorem prover may be a=100000 and y =0. Using this input, the application may now reach the inner branch on line 4. Since 100000 is not less than 2, this branch will not be taken. This means that the path conditions are now a=100000 and a ≥ c, which will be negated to have the other path followed meaning that a<c. The theorem prover will next examine a and b to satisfy a=100000, a<c and x=2b. One example of this may be a=100000 and b=50001. Using these assumptions, the error on line 5 will be reached and all possible paths will have been followed.

While traditional concolic based approaches do offer some benefits in comparison to standard symbolically based methods, the number of possible paths to be explored for each method is still impractically large for most situations. Typically, only small parts of the program state space may be explored (Sen et al., 2005). This is largely because as the length of the executions grow, maintaining and solving symbolic constraints along the execution path become more expensive. Various program paths may be explored exhaustively, however both symbolic and concolic based techniques are ill suited for exploring deep program states which are only reached after long program executions (Majumdar & Sen, 2007).



```
public void Run(int a, int b){
    if(a <b){
        System.out.println("top");
    }else{
         System.out.println("bottom");
    }
}
```

Figure 9. Sample Code.

```
### PCs: 1 1 0
--------original PC------------
original pc # = 1
a_1_SYMINT < b_2_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_1_SYMINT < b_2_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT < b_2_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT < b_2_SYMINT
SPC # = 0
--> # = 1
a_1_SYMINT < b_2_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
top
--------original PC------------
original pc # = 1
a_1_SYMINT >= b_2_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_1_SYMINT >= b_2_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT >= b_2_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT >= b_2_SYMINT
SPC # = 0
--> # = 1
a_1_SYMINT >= b_2_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
Bottom
```

Figure 10. Sample Concolic output.

Figure 9 shows a code snippet that undergoes concolic analysis, which is demonstrated in Figure 10. This was generated using a modified version of Java Path

Finder (JPF). Path conditions, along with all possible branches and paths the application may take are displayed in this resulting output. In addition to software testing, concolic analysis has traditionally been used in several other areas. Some of which includes the generation of test input data and fault localization (Artzi, Dolby, Tip, & Pistoia, 2010; Wassermann et al., 2008).

In order to traverse the paths of an application, concolic analysis uses a depth-first search. A depth-first search is a way of exploring all possible paths of a tree by starting at the root and traversing each possible branch as far as possible. Once a path has been fully examined, the search will investigate the next branch until it reaches a terminal point (Li & Garcia-Luna-Aceves, 2007; Sibeyn, Abello, & Meyer, 2002; Tatti & Cule, 2011). This is a rather important process in concolic analysis for several reasons. Too short of a search means that not enough paths will be explored. Too deep of an analysis may lead to an extremely large or time consuming exploration of the program space. In the event an infinite loop is encountered, the tree may be impossible to fully traverse (Majumdar & Sen, 2007; Sen, 2007). Optimally, a middle ground will be found that offers an adequate exploration of an adequate number of execution spaces, but does not take an unreasonable number of paths. There are several different methods for handling this problem. The traditional approach is to backtrack in order to define the search depth. However, for large or complex application segments, this is still a very expensive task. Recent research has been done in order to make this analysis into a more efficient process (Sen et al., 2005).

One method of reducing the possible negative impacts of a depth-first search is to use a bounded or depth-limited search. This alternative will explore a tree in the same manner as a depth-first search, but will be merely limited by the maximum depth limit which may be traversed. The main benefit to this method is that infinitely deep paths will never be explored (Bardin & Herrmann, 2009). A disadvantage of this method will be that the tree is not explored beneath the defined level of the defined limit value. This leads to the chance that the entire tree will not be analyzed (Bond, Srivastava, McKinley, & Shmatikov, 2010; D. Yang & Powers, 2005). Appendix B contains examples of the concolic output of Figure 9 with a depth search of 1, 3 and 5.

Lakhotia. Harman, & McMinn (2008) describe a concolic search process where they set the depth search parameter to infinite. They describe how their mechanism got caught in an infinite loop numerous times and often needed a large number of iterations to complete its search. This is often the case with unbounded depth searches such as this. Other works discuss possible alternative methods for resolving this path exploration issue, but no definitive solution appears to have been discovered to best serve every possible situation uniformly (Bardin & Herrmann, 2009; Lakhotia et al., 2008; Majumdar & Sen, 2007; Sen, 2007).

Chapter 3

Methodology

Overview of Research Methodology

In order to detect code clone candidates, CCCD performs concolic analysis which analyzes the flow or paths taken by the application. The detection process is comprised of two primary phases. The first is to run concolic analysis on the source code. A modified version of an existing tool known as Java Path Finder (JPF) is used to perform this step[2]. The generated concolic output is then examined for code clone candidates by looking for repeated or like segments in this output. Figure 11 depicts the components of CCCD and the necessary steps to discover code clone candidates. This current research only represents a proof of concept. Further work may be done in order to make this into a complete tool.

[2] http://babelfish.arc.nasa.gov/trac/jpf

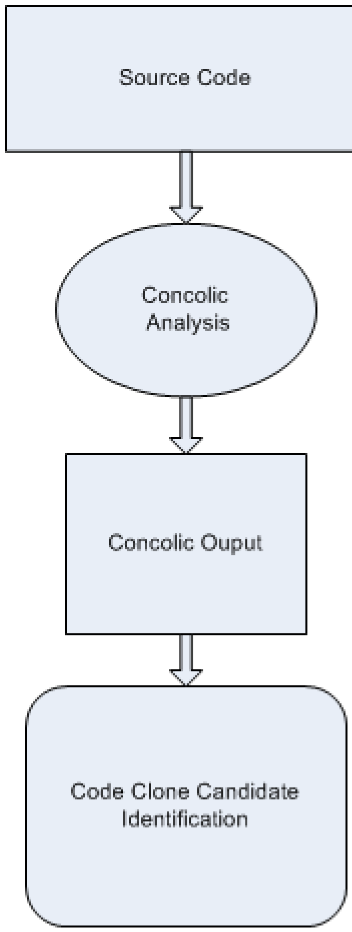
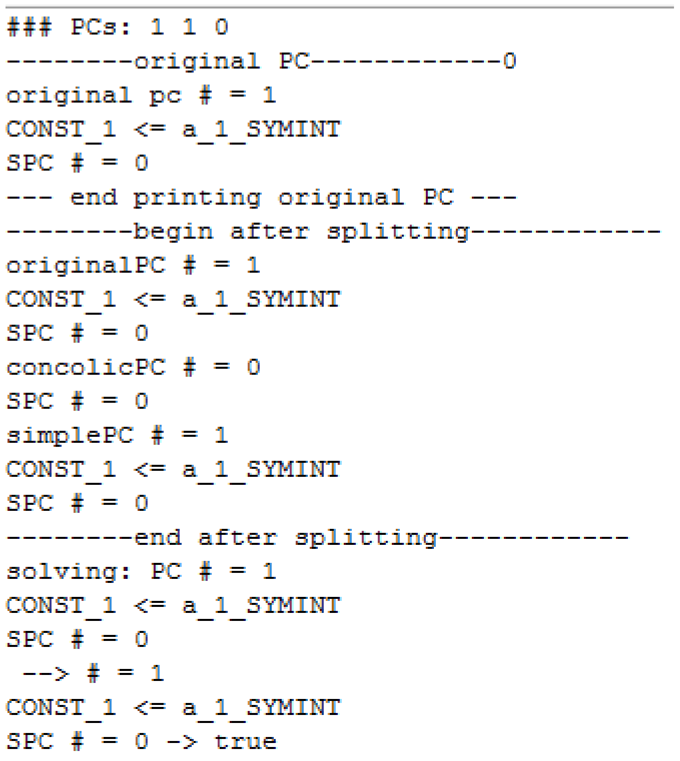


Figure 11. CCCD Sequence and Flow

Concolic Analysis Generation

In order to generate the necessary concolic output, a modified version JPF is used. This is a free application which has been utilized in previous research (Ihantola, 2006; Kalibera, Parizek, Malohlava, & Schoeberl, 2010; Visser, Pasareanu, & Khurshid, 2004). JPF was also chosen since it is a robust tool that is easy to use, configure and modify. Its availability also means that the CCCD process is repeatable for other researchers.



```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST_1 <= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST_1 <= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST_1 <= a_1_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST_1 <= a_1_SYMINT
SPC # = 0
 --> # = 1
CONST_1 <= a_1_SYMINT
SPC # = 0 -> true
```

Figure 12. JPF example concolic output.

Running JPF against the source code of an application is a relatively simple process. The source code of the desired software may be analyzed by this tool through Eclipse. Figure 12 shows a simple example of concolic output using JPF. Ultimately, numerous sets of concolic analysis are generated. Since concolic analysis only cares about the flow of an application and not the precise syntax of the source code, no normalization is expected to be required (Sen, 2007).

JPF requires numerous modifications and configuration changes in order to make it into a functional component of CCCD. While the core concolic engine was not altered, some of its output was changed. The main concolic engine was not modified for several reasons. The first is that concolic analysis by itself is largely capable of discovering clones. Subsequent alterations to its output were simply needed to make the clone discovery process more robust and effective. Additionally, CCCD is largely agnostic to the exact concolic analysis tool implemented. An existing concolic analysis tool should be interchangeable into the CCCD process.

The alterations made to JPF were made through the application's listener and configuration files. These were created as part of JPF in order to make it easily configurable by other developers. Some of the changes that were made to the listener are the removal of various unneeded output variables such as the concolic counters and other configuration settings which are output as part of the concolic generation process.

A significant hurdle that needed to be overcome was the selection of the proper depth search level for concolic analysis to use in order to discover code clones. Analyzing too few paths will yield too few results, while examining too many will lead to too many paths being analyzed which could lead to an overly complex or time consuming exploration of program space (Bardin & Herrmann, 2009; Lakhotia et al., 2008).

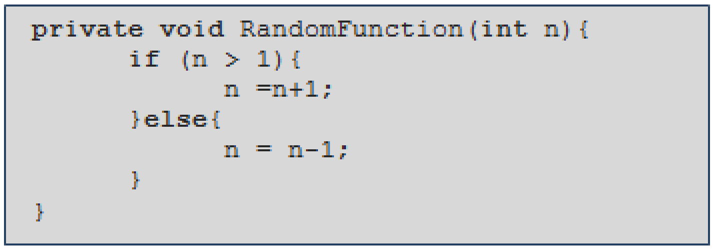
This is a serious matter for CCCD. Too small of a depth search means that not enough concolic results will be returned since not enough paths will be explored. Searching too many paths could lead to infinite loops being encountered which will effectively stop the concolic analysis process. Additionally, if there is a point where a specific search depth will not create a more accurate process, then traversing any more paths simply represents wasted resources.

Concolic analysis has the ability to use lazy instantiation in this decision making process. This means that the components of the method inputs are created in an on demand basis. Input sizes do not require a priori bounding (Khurshid et al., 2003). The decision to use lazy instantiation was an important one for ensuring the quality of CCCD.

Using lazy initialization with concolic values, the execution tree of the application was generated. If this tree is infinite, this approach discovers all of the possible nodes of the tree. This means that a test set with maximum test coverage is created (Ihantola, 2006). Ensuring maximum coverage is of significance importance in detecting clones. If all paths are not appropriately explored, this may affect the concolic output and lead to incorrect determinations during the clone identification process.

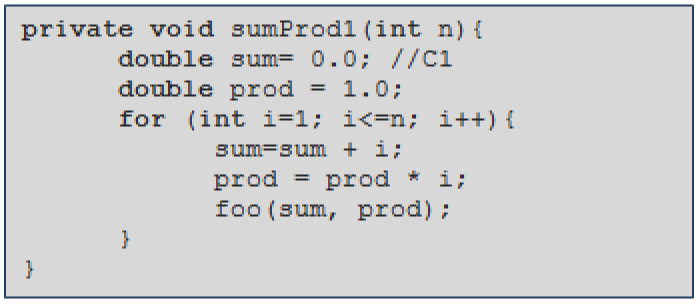
Code Clone Candidate Identification

The concolic output is then examined for identical or repeated portions. Sections are compared using the noted start and end of each method, so that specific code blocks may be searched for. Since concolic execution only cares about the concolic path or functionality of the application being examined, duplicate output represents a clone candidate. This process is done using a *diff* tool to look for repeated segments. The concolic output of the examined methods is then compared and exact matches will identify a clone candidate.



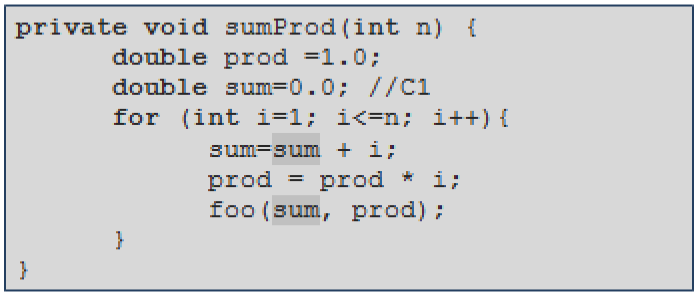
```
private void RandomFunction(int n){
      if (n > 1){
            n =n+1;
      }else{
            n = n-1;
      }
}
```

Figure 13. Example function.



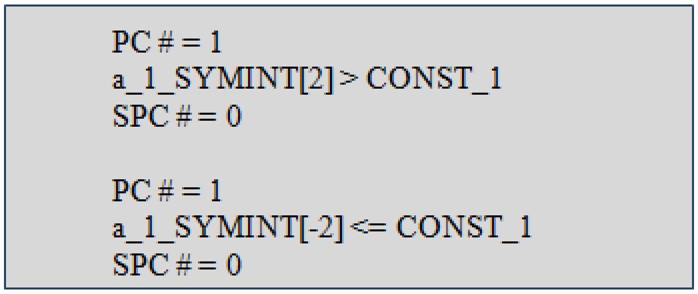
```
private void sumProd1(int n){
      double sum= 0.0; //C1
      double prod = 1.0;
      for (int i=1; i<=n; i++){
            sum=sum + i;
            prod = prod * i;
            foo(sum, prod);
      }
}
```

Figure 14. Example clone function.



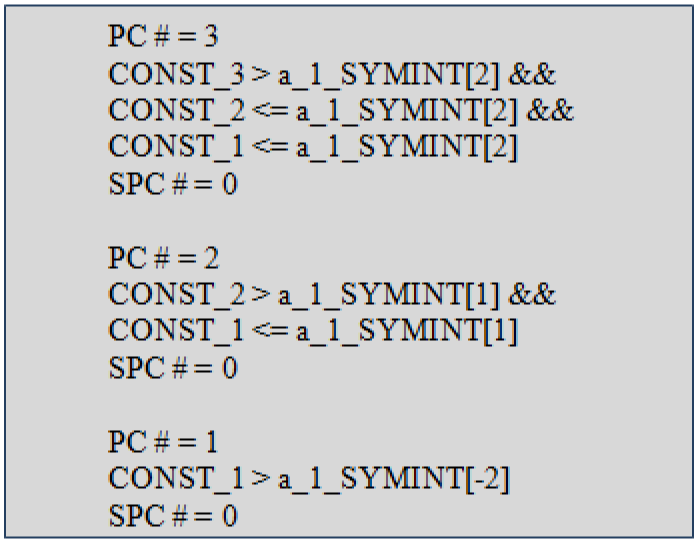
```
private void sumProd(int n) {
      double prod =1.0;
      double sum=0.0; //C1
      for (int i=1; i<=n; i++){
            sum=sum + i;
            prod = prod * i;
            foo(sum, prod);
      }
}
```

Figure 15. Example second clone function.



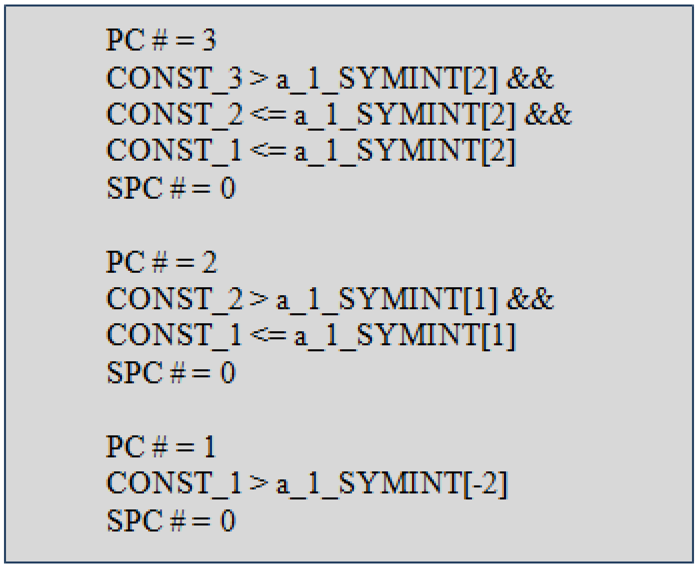
```
PC # = 1
a_1_SYMINT[2] > CONST_1
SPC # = 0

PC # = 1
a_1_SYMINT[-2] <= CONST_1
SPC # = 0
```

Figure 16. Example concolic output of Figure 13.



```
PC # = 3
CONST_3 > a_1_SYMINT[2] &&
CONST_2 <= a_1_SYMINT[2] &&
CONST_1 <= a_1_SYMINT[2]
SPC # = 0

PC # = 2
CONST_2 > a_1_SYMINT[1] &&
CONST_1 <= a_1_SYMINT[1]
SPC # = 0

PC # = 1
CONST_1 > a_1_SYMINT[-2]
SPC # = 0
```

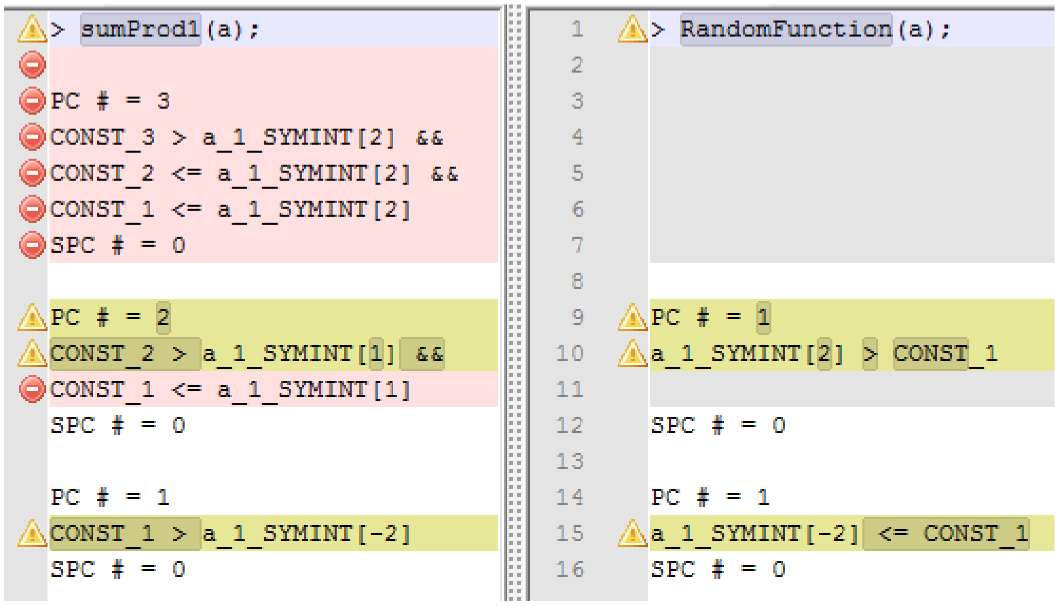
Figure 17. Concolic output of Figure 14**.**



```
PC # = 3
CONST_3 > a_1_SYMINT[2] &&
CONST_2 <= a_1_SYMINT[2] &&
CONST_1 <= a_1_SYMINT[2]
SPC # = 0

PC # = 2
CONST_2 > a_1_SYMINT[1] &&
CONST_1 <= a_1_SYMINT[1]
SPC # = 0

PC # = 1
CONST_1 > a_1_SYMINT[-2]
SPC # = 0
```

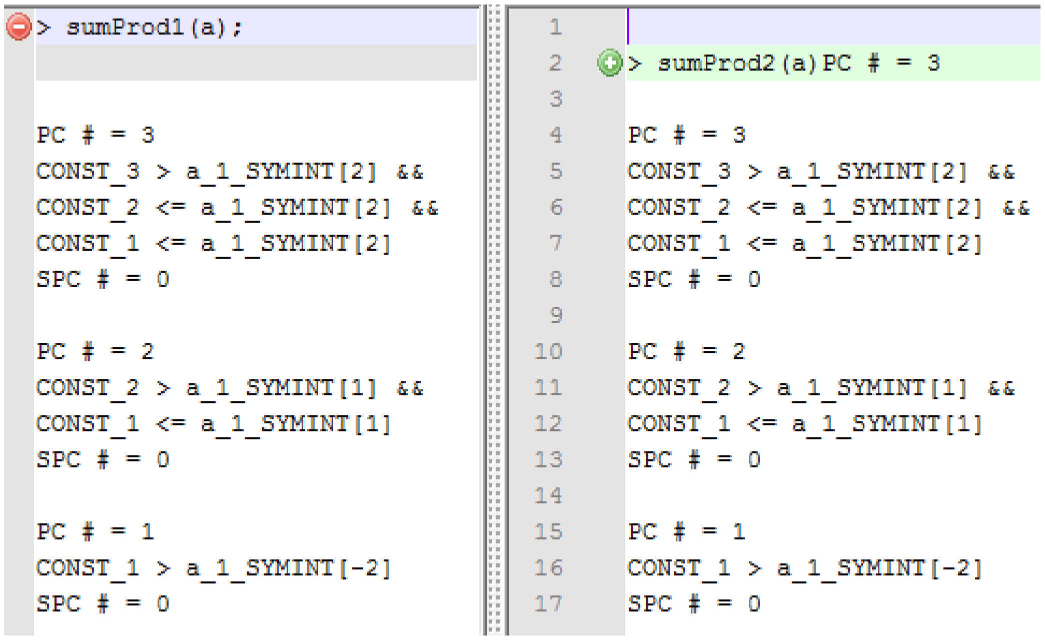
Figure 18. Concolic output of Figure 15.

Figure 16, Figure 17 and Figure 18 represent the concolic output of the code snippets. Figure 13 represents a method with distinct functionality and is not a clone (Roy et al., 2009).



```
> sumProd1(a);                              > RandomFunction(a);

PC # = 3
CONST_3 > a_1_SYMINT[2] &&
CONST_2 <= a_1_SYMINT[2] &&
CONST_1 <= a_1_SYMINT[2]
SPC # = 0

PC # = 2                                    PC # = 1
CONST_2 > a_1_SYMINT[1] &&                  a_1_SYMINT[2] > CONST_1
CONST_1 <= a_1_SYMINT[1]
SPC # = 0                                   SPC # = 0

PC # = 1                                    PC # = 1
CONST_1 > a_1_SYMINT[-2]                    a_1_SYMINT[-2] <= CONST_1
SPC # = 0                                   SPC # = 0
```

Figure 19. Comparison of concolic output of Figure 16 and Figure 17.



```
> sumProd1(a);
                                            > sumProd2(a)PC # = 3

PC # = 3                                    PC # = 3
CONST_3 > a_1_SYMINT[2] &&                  CONST_3 > a_1_SYMINT[2] &&
CONST_2 <= a_1_SYMINT[2] &&                 CONST_2 <= a_1_SYMINT[2] &&
CONST_1 <= a_1_SYMINT[2]                    CONST_1 <= a_1_SYMINT[2]
SPC # = 0                                   SPC # = 0

PC # = 2                                    PC # = 2
CONST_2 > a_1_SYMINT[1] &&                  CONST_2 > a_1_SYMINT[1] &&
CONST_1 <= a_1_SYMINT[1]                    CONST_1 <= a_1_SYMINT[1]
SPC # = 0                                   SPC # = 0

PC # = 1                                    PC # = 1
CONST_1 > a_1_SYMINT[-2]                    CONST_1 > a_1_SYMINT[-2]
SPC # = 0                                   SPC # = 0
```

Figure 20. Comparison of concolic output of Figure 15 and Figure 16

Figure 20 represents a simple *diff* done using Notepad++[3]. Since the analyzed source code (Figure 14 & Figure 15) are dissimilar, the *diff* performed on the concolic data shows significant differences between this output. However, since Figure 14 and Figure 15 represent code clones, a diff done on their concolic output in Figure 20 shows that the concolic output is similar. This dissertation represents a proof of concept. Future work may be done in order to make this comparison method into an automatic process and eliminate much of the need for human interaction.

Specific Research Methods Employed

The research methods employed attempted to find clones of all four categories. Once the output from CCCD was completed, a manual process was used to determine the accuracy of the clone candidate discovery process. In order to validate the proposed process, each of the four clone types were injected into the system. These were taken from previous research (Krawitz, 2012; Roy et al., 2009). The output of CCCD was then manually checked to ensure that it properly detected all four of these classes of clones. This validation effort closely mimics that performed by Krawitz (2012) on this tool.

CCCD then analyzed several open source applications which acted as benchmarks. Since JPF is only compatible with software written in Java, all selected applications were written in this language. The proposed method is only limited by the chosen concolic analysis system and is language independent as a general process. The

[3] http://notepad-plus-plus.org/

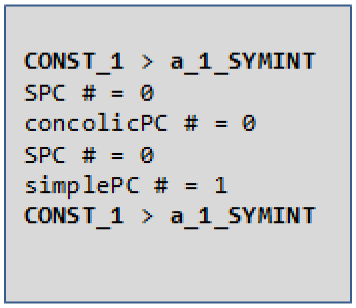
applications used as benchmarks were JDraw[4], DrJava[5], JabRef[6], Jrand[7], Tuxguitar[8] and RES[9]. All of these applications are open source and are freely available to the public. When this data was further processed by CCCD, it demonstrated the ability of concolic analysis to detect code clone candidates.

Instrument Development and Validation

CCCD is comprised of two existing tools, JPF and an application for performing *diffs*, which in this case will be Notepad++. The initial setup of JPF was a relatively easy and straightforward process which was accomplished by following instructions on the application's website[10]. JPF was then significantly modified. These alterations included functionality changes largely implemented through the application's listener and configuration modifications. While the output given to Notepad++ had to be altered, this *diff* application did not.

Once JPF had been setup and the proper instructions were followed in order to allow for basic concolic analysis to be performed, further alterations were then required in order to make JPF a functional component of CCCD. The largest modification made was the alteration of the listener. This is customized software in JPF which changes the desired functionality of the application. Modifying this listener gives the ability to customize virtually any of the functionality in JPF.

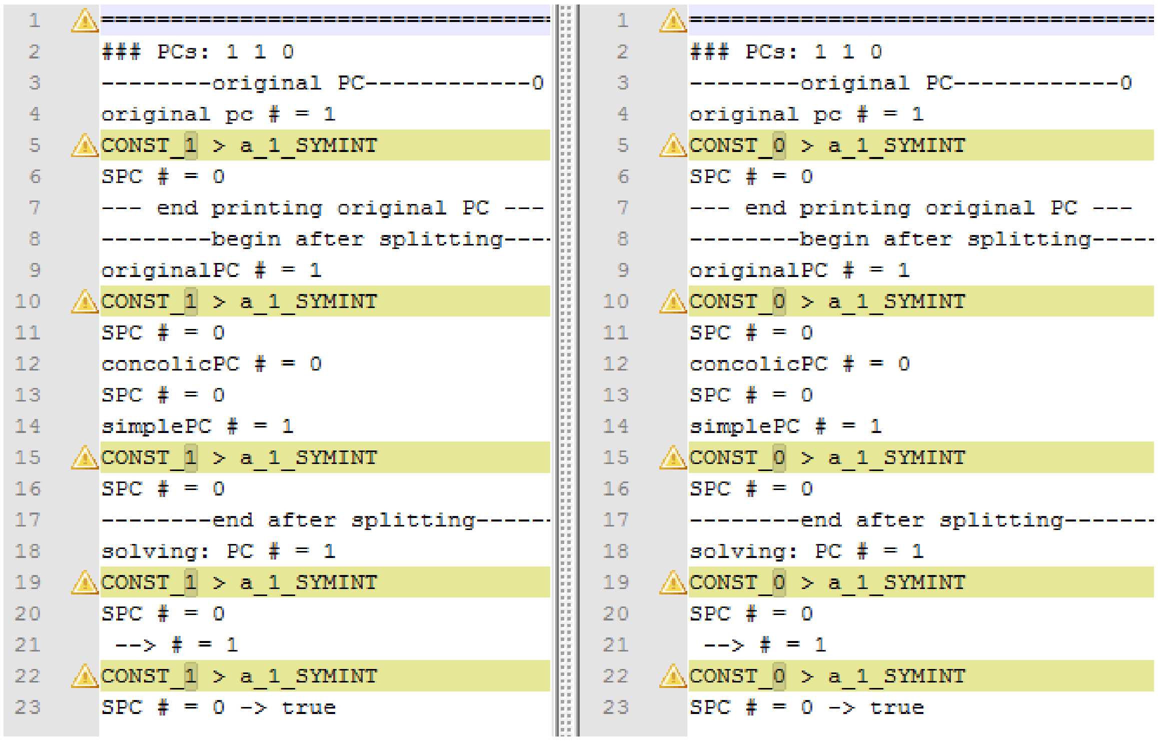
[4] http://jdraw.sourceforge.net/
[5] http://drjava.sourceforge.net/
[6] http://jabref.sourceforge.net/
[7] http://sourceforge.net/projects/jrand/
[8] http://sourceforge.net/projects/tuxguitar/
[9] http://sourceforge.net/projects/opencobol2java/
[10] http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/start



```
CONST_1 > a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST_1 > a_1_SYMINT
```

Figure 21. Example of concolic counters.

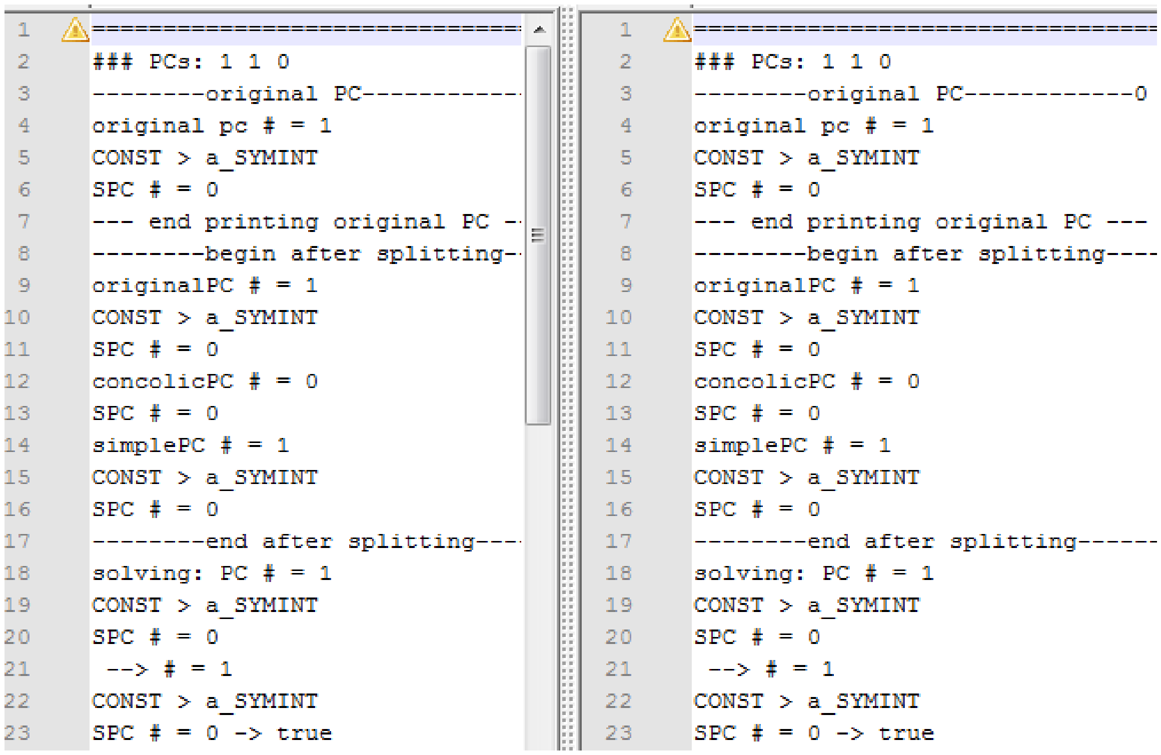
While these modifications to JPF significantly enhanced the concolic values returned as part of the clone candidate identification process, some cleansing was required. The first step was to remove the variable counters from the concolic output. These are merely numeric assignments assigned to variables by JPF in an incremental fashion. They have no effect on the actual flow of the application and no benefits to the clone detection process. An example of these counter values are displayed in Figure 21 and are shown as "CONST_1" and "a_1_SYMINT" in bold.



```
====================================
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST_1 > a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting---
originalPC # = 1
CONST_1 > a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST_1 > a_1_SYMINT
SPC # = 0
--------end after splitting------
solving: PC # = 1
CONST_1 > a_1_SYMINT
SPC # = 0
 --> # = 1
CONST_1 > a_1_SYMINT
SPC # = 0 -> true
```

```
====================================
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST_0 > a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting-----
originalPC # = 1
CONST_0 > a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST_0 > a_1_SYMINT
SPC # = 0
--------end after splitting-------
solving: PC # = 1
CONST_0 > a_1_SYMINT
SPC # = 0
 --> # = 1
CONST_0 > a_1_SYMINT
SPC # = 0 -> true
```

Figure 22. Example variable count difference before cleaning.

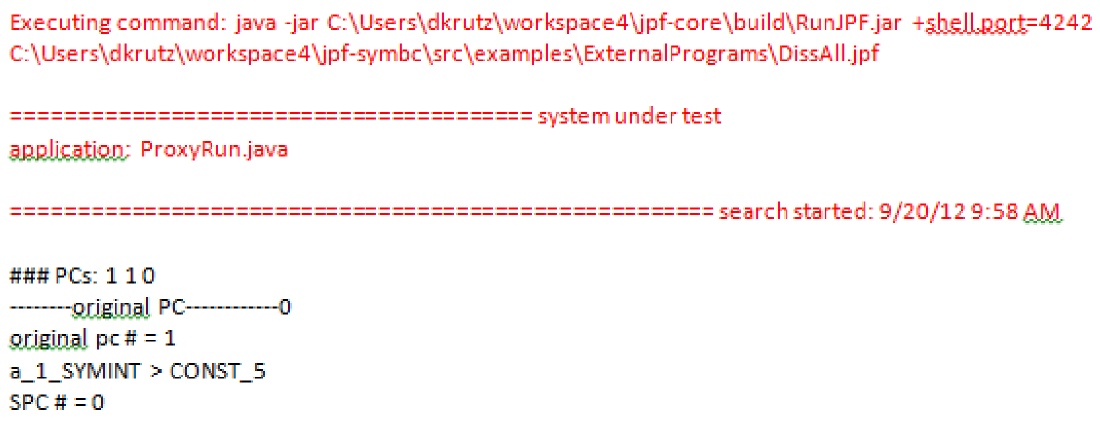


```
================================
### PCs: 1 1 0
--------original PC-----------
original pc # = 1
CONST > a_SYMINT
SPC # = 0
--- end printing original PC -
--------begin after splitting-
originalPC # = 1
CONST > a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST > a_SYMINT
SPC # = 0
--------end after splitting---
solving: PC # = 1
CONST > a_SYMINT
SPC # = 0
 --> # = 1
CONST > a_SYMINT
SPC # = 0 -> true
```

```
=================================
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST > a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting----
originalPC # = 1
CONST > a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST > a_SYMINT
SPC # = 0
--------end after splitting------
solving: PC # = 1
CONST > a_SYMINT
SPC # = 0
 --> # = 1
CONST > a_SYMINT
SPC # = 0 -> true
```

Figure 23. Example variable count difference after cleaning.

Figure 22 represents a *diff* on some example concolic output before the cleaning process has taken place and the integer values have been removed. Figure 23 shows these same two code segments after the cleaning process has occurred and the two segments are now found to be identical. This cleaning process is important for removing minor differences between code segments that are simply a byproduct of JPF. These are not at all indicative of actual differences between the functionality of two segments of code. This cleaning process took place by altering the JPF's listener. Once this adjustment occurred, the modified concolic output was then automatically displayed.

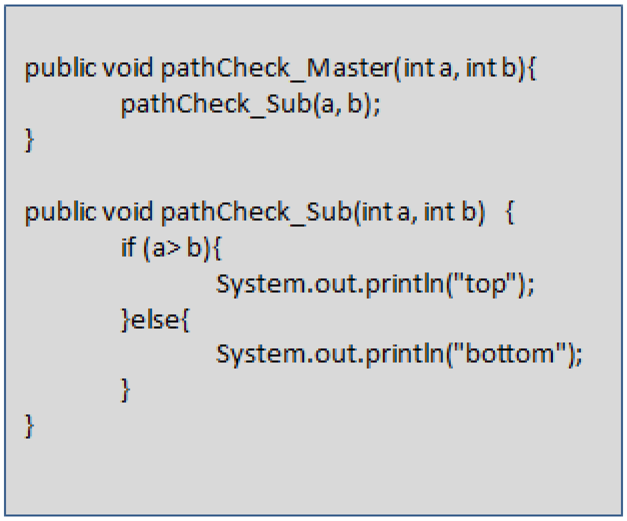


```
Executing command: java -jar C:\Users\dkrutz\workspace4\jpf-core\build\RunJPF.jar +shell.port=4242
C:\Users\dkrutz\workspace4\jpf-symbc\src\examples\ExternalPrograms\DissAll.jpf

===================================== system under test
application: ProxyRun.java

==================================================== search started: 9/20/12 9:58 AM

### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_1_SYMINT > CONST_5
SPC # = 0
```

Figure 24. Information to be removed from concolic output.

Several other pieces of information that are a byproduct of JPF were then removed from the output. Figure 24 represents the beginning of a concolic output file. In this example, several pieces of information exist which are not needed for clone discovery using concolic analysis. This information is highlighted in red. As part of the CCCD process, this output was removed since it has no bearing on the functionality of the examined source code and is merely a byproduct of concolic analysis.

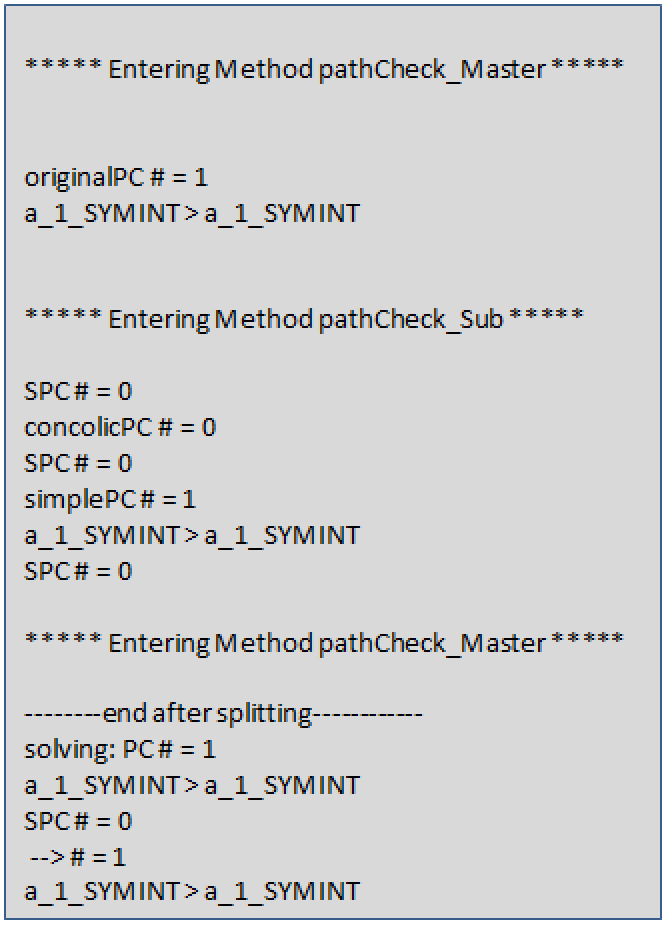
There are numerous settings changes that had to be modified in JPF in order to use it as part of the CCCD process. While these alterations were not difficult to implement, they did require a substantial amount of thought in determining what they should be and how they will interact with the clone detection process. One of these settings is the debug flag, which should be enabled. This instructs JPF to output the appropriate concolic data for analysis. Not having this option enacted will not produce the appropriate concolic output for examination.



```
public void pathCheck_Master(int a, int b){
        pathCheck_Sub(a, b);
}

public void pathCheck_Sub(int a, int b)   {
        if (a> b){
                System.out.println("top");
        }else{
                System.out.println("bottom");
        }
}
```

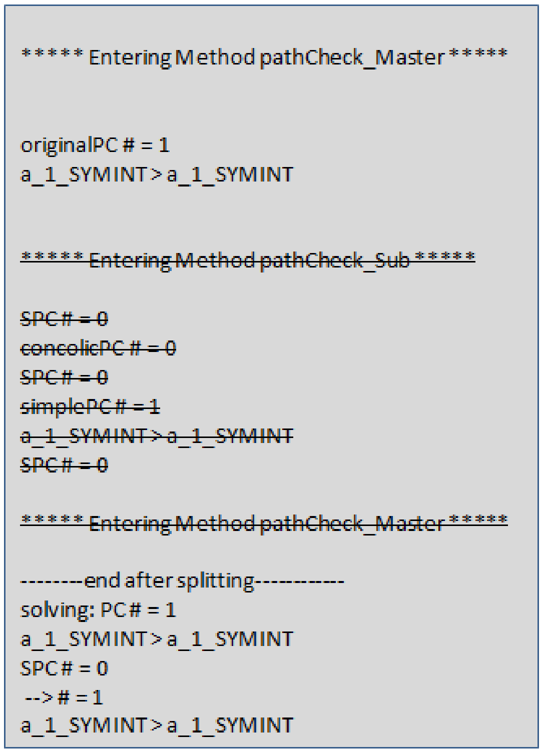
Figure 25. Example of method referencing another method.

Since the methods may reference and call other functions in the application, the path explored by concolic analysis may traverse multiple methods. Since the goal of this work is to only find clones at the method level, the paths which enter outside functions must be identified. Figure 25 represents a simple method “pathCheck_Master” which references another method “pathCheck_Sub”. If the modified version of JPF is run against pathCheck_Master, its flow will enter “pathCheck_Sub”. The concolic output will not differentiate between the two methods.



```
***** Entering Method pathCheck_Master *****


originalPC # = 1
a_1_SYMINT > a_1_SYMINT


***** Entering Method pathCheck_Sub *****

SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT > a_1_SYMINT
SPC # = 0

***** Entering Method pathCheck_Master *****

--------end after splitting-----------
solving: PC # = 1
a_1_SYMINT > a_1_SYMINT
SPC # = 0
 --> # = 1
a_1_SYMINT > a_1_SYMINT
```

Figure 26. Modified concolic output to indicate different method.



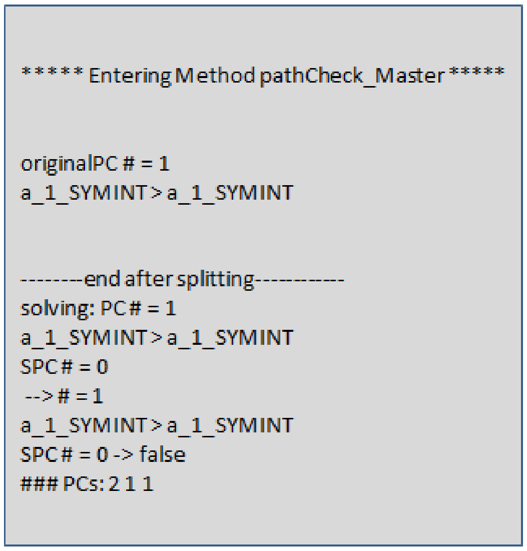
```
***** Entering Method pathCheck_Master *****

originalPC # = 1
a_1_SYMINT > a_1_SYMINT
```

~~***** Entering Method pathCheck_Sub *****~~

~~SPC# = 0~~
~~concolicPC # = 0~~
~~SPC# = 0~~
~~simplePC# = 1~~
~~a_1_SYMINT > a_1_SYMINT~~
~~SPC# = 0~~

~~***** Entering Method pathCheck_Master *****~~

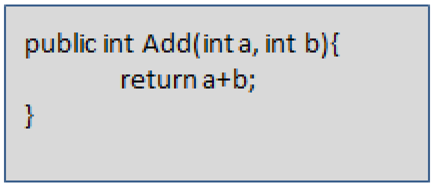
```
--------end after splitting-----------
solving: PC# = 1
a_1_SYMINT > a_1_SYMINT
SPC# = 0
 --> # = 1
a_1_SYMINT > a_1_SYMINT
```



```
***** Entering Method pathCheck_Master *****

originalPC # = 1
a_1_SYMINT > a_1_SYMINT

--------end after splitting-----------
solving: PC# = 1
a_1_SYMINT > a_1_SYMINT
SPC# = 0
 --> # = 1
a_1_SYMINT > a_1_SYMINT
SPC# = 0 -> false
### PCs: 2 1 1
```

Figure 27. External function information removed.

A modification to the listener was required to add the ability to notate that a different method is being referenced. Figure 26 represents the concolic output where a statement has been added with the name of the new method which has been entered by the concolic analysis. The modified listener then has the capability to remove information from the noted paths. This idea is represented in Figure 27. It is important to note that even though the method names are added to the concolic output, they are not specifically used in the comparison process used to detect clones. This means that CCCD’s ability to detect type-3 and type-4 clones is not affected in any way.



```
public int Add(int a, int b){
        return a+b;
}
```

Figure 28. Example method to be targeted.

CCCD targets individual methods by specifying them in the application's configuration file. Focusing on a specific method is done by using the syntax "*symbolic.method = ClassName.MethodName(sym#sym).*" ClassName represents the class being targeted while MethodName is the respective method. The values "sym#sym" is used to indicate two inputs into the method signature. For example, if the code in Figure 28 was to be analyzed and was located in a class called "Calculator", the configuration file would need to contain "*symbolic.method = Calculator.Add(sym#sym)"* Each function which is to be analyzed needs to be referenced in this manner. Future research may be done in order to add the functionality to automatically execute all methods of the application in a single command. However, that is out of the scope of this current research and does not affect the question of if concolic analysis can be used to detect clone candidates.

Once these alterations have been performed and the output has been cleaned, a simple *diff* was performed on it. This may be done using an existing tool, which in this case was Notepad++. The same process was then used on the next method to be compared. Ultimately, two sets of concolic output exist and a simple *diff* process using Notepad++ was then done in order to indicate a possible clone candidate.

Resource Requirements

The resources required to complete this research were not an issue. A concolic testing application was required to perform concolic analysis on the target application. This concolic generation application was altered in order to be an effective component of CCCD. JPF was able to meet these requirements. Additionally, a tool capable of performing a diff on the modified output was required. This research used Notepad++.

Summary

This research used several open source applications as benchmarks for demonstrating the effectiveness of CCCD in discovering code clones. Existing clones were first identified in the benchmark applications. CCCD was then run to see how many of these clones it was able to discover. Clones of all four types as identified by previous research were then inserted into these benchmark applications. CCCD was then run to see how many of these clones it was able to discover.

This work used a modified version of JPF in order to execute concolic analysis on the examined applications. Notepad++ was then used to perform a *diff* on the generated output in order to discover clones. Similar output was indicative of a code clone candidate. Each of these primary components of CCCD are freely available to download.

Chapter 4

Results

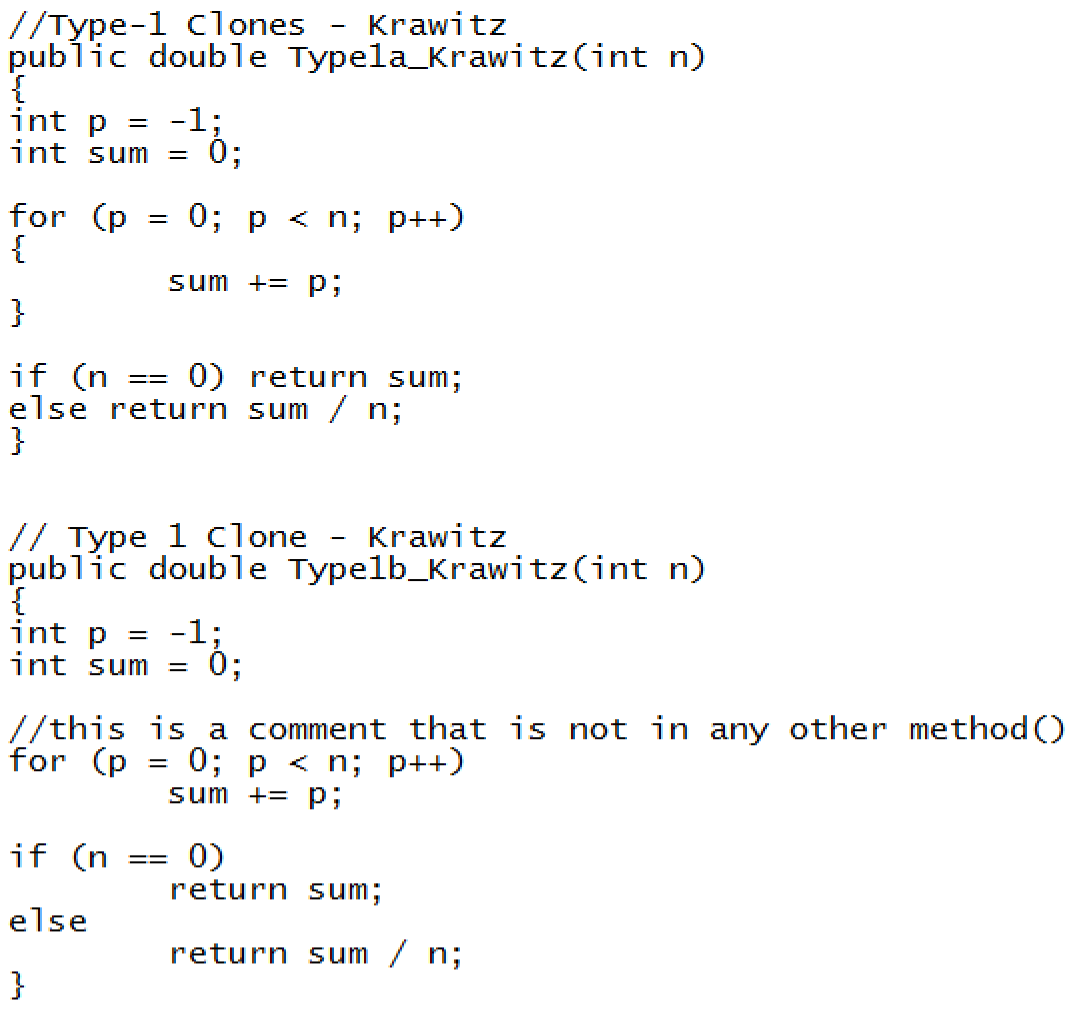
Results Introduction

This dissertation introduced CCCD, a candidate code clone detection method which is based on concolic analysis. This technique was able to detect all four types of code clone candidates. This chapter presents the results and observations found from conducting the experiments described in the previous chapter. Several open source applications were selected and examined for code clone candidates. Additionally, code clones which were defined by previous works were injected into these applications (Krawitz, 2012; Roy et al., 2009). These added clones were helpful in showing that CCCD was capable of discovering clone candidates of all four types. CCCD was then run against these applications and the results were recorded.

CCCD discovered clones on a method level by first performing concolic analysis on each of the desired functions. This output was then compared with the concolic values from other methods in the application. Identical concolic products indicated a clone candidate. Based on the results, CCCD was able to discover the vast majority of clone candidates and only struggled with a single example due various technical limitations of the selected modified concolic analysis tool, JPF. Future development of this tool or the selection of a different concolic analysis application would likely alleviate this issue. This hurdle was by no means a limitation of CCCD, but only of the selected concolic analysis tool.

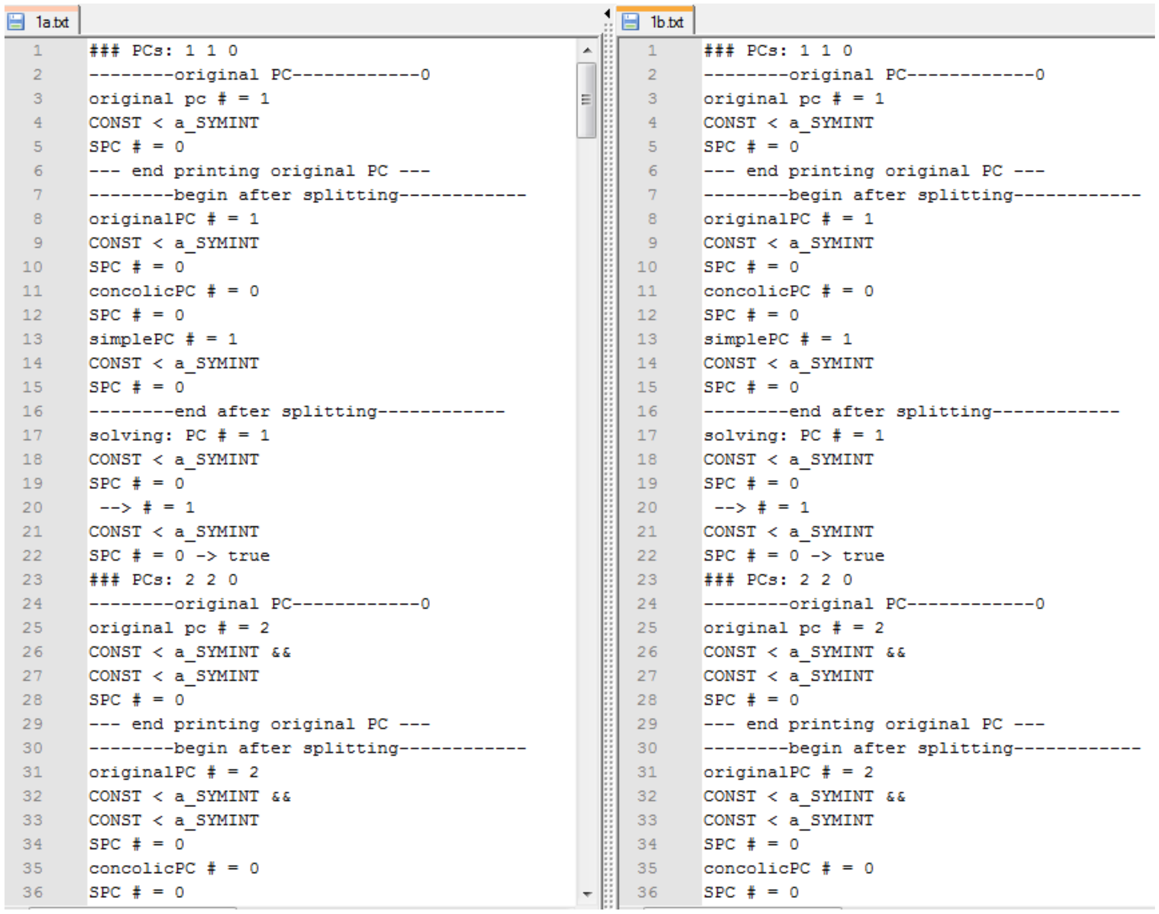
CCCD Results

CCCD was first evaluated against clones which were defined by Krawitz (2012) and Roy (2009). These are shown in Appendix A. These existing clone examples were selected for several reasons. The first is that using them gave an initial indication of the effectiveness of CCCD and what clones and clone types it was able to discover. Secondly, these examples represent a full spectrum of all clone types providing a solid evaluation benchmark for CCCD. Finally, since this information originated from existing research it is repeatable for future analysis. In order to evaluate CCCD against these examples, each of the selected clones by Roy (2009) and Krawitz (2012) were analyzed by CCCD. Their output was then compared and the determination was made if a clone was successfully identified.



```
//Type-1 Clones - Krawitz
public double Type1a_Krawitz(int n)
{
int p = -1;
int sum = 0;

for (p = 0; p < n; p++)
{
        sum += p;
}

if (n == 0) return sum;
else return sum / n;
}


// Type 1 Clone - Krawitz
public double Type1b_Krawitz(int n)
{
int p = -1;
int sum = 0;

//this is a comment that is not in any other method()
for (p = 0; p < n; p++)
        sum += p;

if (n == 0)
        return sum;
else
        return sum / n;
}
```

Figure 29. Type-1 Clones in Injected Code.



1a.txt

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
```

1b.txt

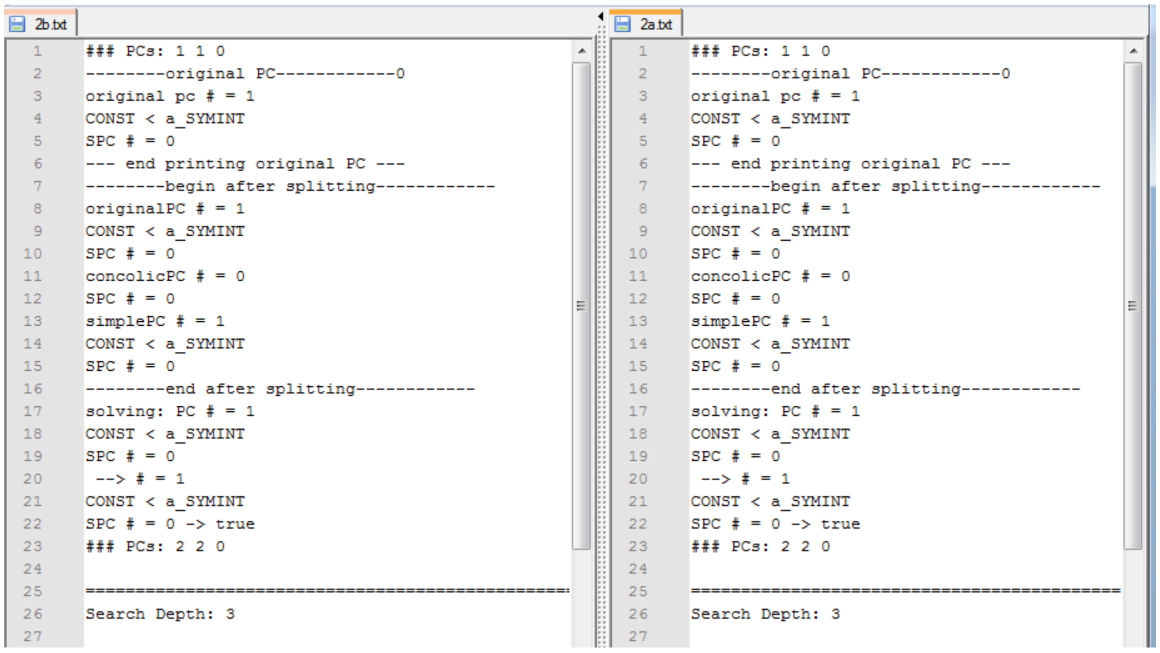
```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
```

Figure 30. Type-1 Concolic Output of Figure 29.

An example of an inserted type-1 clone is represented in Figure 29. These clones are semantically identical (Koschke, 2007). CCCD was able to detect the existence of a code clone candidate between these two functions by comparing the concolic output of these two methods. These results are shown in Figure 30. The concolic output of both of these methods is identical, thus correctly indicating a code clone. Due to the significant length of the concolic output, an abbreviated segment shown in Figure 30. Complete results for all injected clones may be found in Appendix C.

```
//Type-2 Clones - Krawitz
public double Type2a_Krawitz(int n){
        int q = -1;
        double sum = 0;

        for (q = 0; q < n; q++){
                sum += q;
        }

        if (n == 0) return sum;
        else return sum / n;
}

//Type-2 Clones - Krawitz
        public double Type2b_Krawitz(int t){
        int p = -1;
        int tot = 0;

        //this is a comment that is not the same as any other comment
        for (p = 0; p < t; p++)
                tot += p;

                if (t == 0)
                        return tot;
                else
                        return tot / t;
}
```

Figure 31. Type-2 Clones in Injected Code.


2b.txt

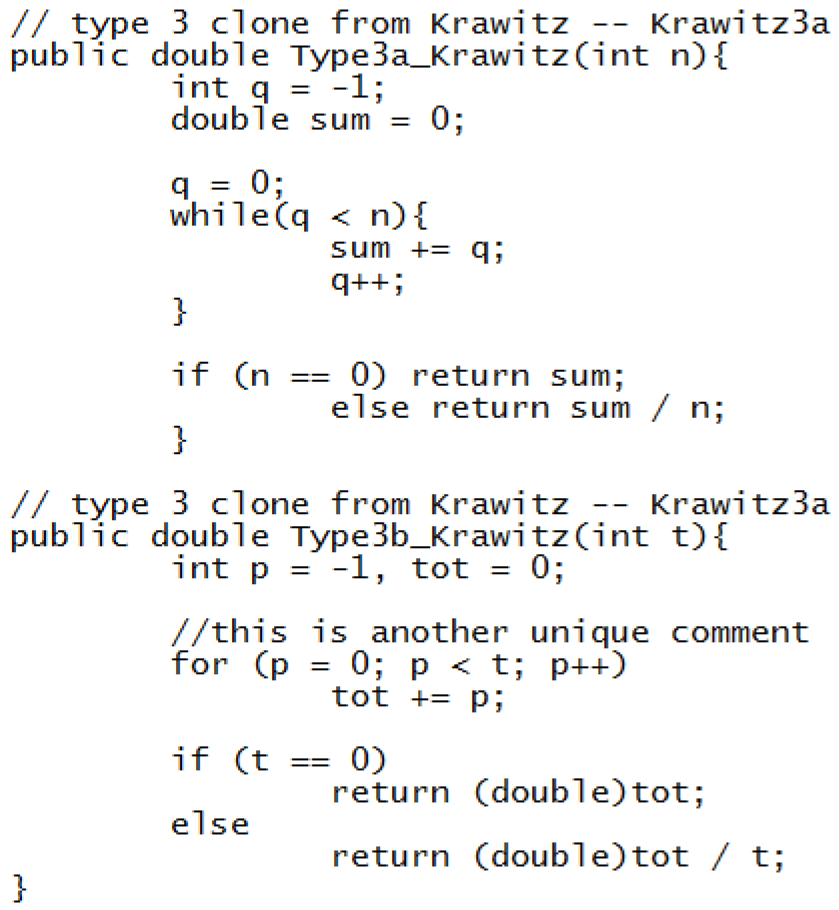
```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
---------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0

==================================================
Search Depth: 3
```

2a.txt

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
---------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0

==============================================
Search Depth: 3
```

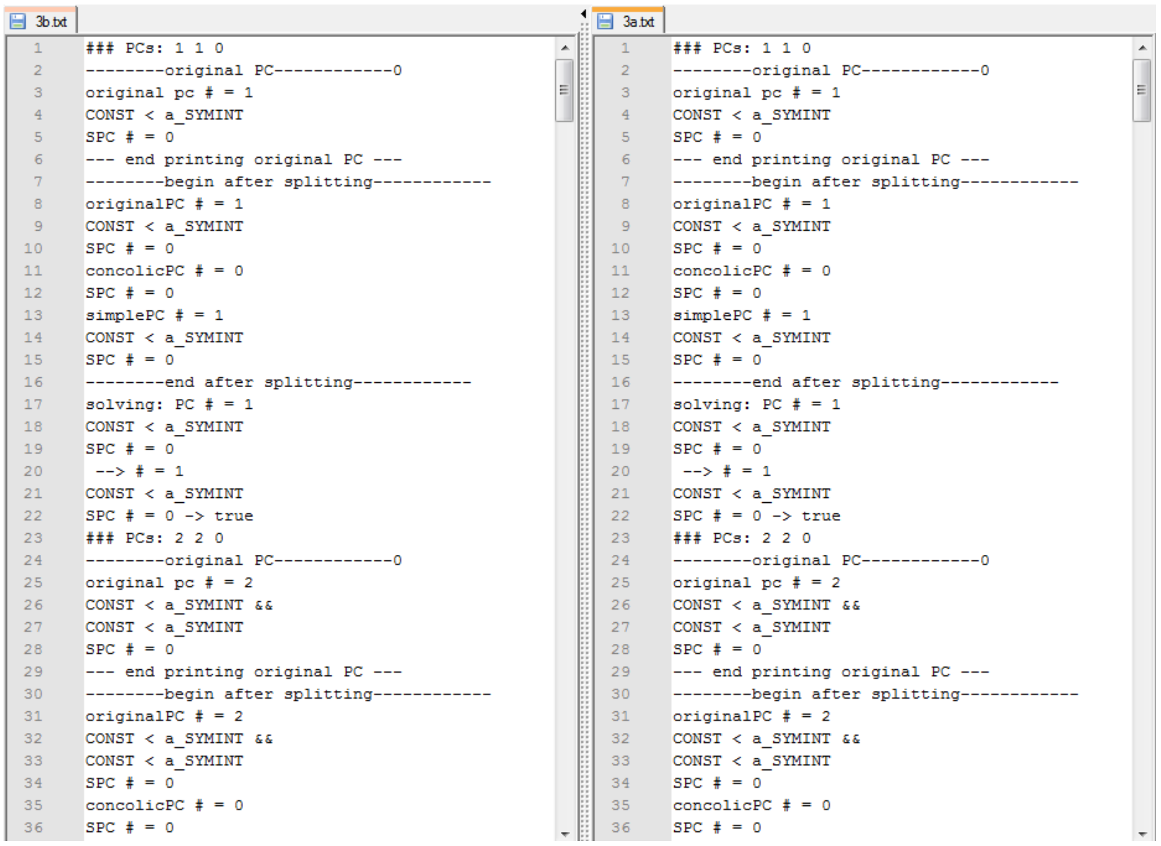
Figure 32. Type-2 Concolic Output of Figure 31.

Type-2 clones are syntactically identical except for variable identifiers and variable types (Koschke, 2007). Figure 31 represents an example of a type-2 clone as defined by Krawitz (2012). Figure 32 displays the concolic output created by CCCD for each of the analyzed methods. Based on the *diff* process, there are no differences between the output generated by CCCD for these methods. This is a correct indication of a code clone. Due to the significant length of the concolic output, an abbreviated segment is shown in Figure 32. Complete results for all injected clones may be found in Appendix C.



```
// type 3 clone from Krawitz -- Krawitz3a
public double Type3a_Krawitz(int n){
        int q = -1;
        double sum = 0;

        q = 0;
        while(q < n){
                sum += q;
                q++;
        }

        if (n == 0) return sum;
                else return sum / n;
        }

// type 3 clone from Krawitz -- Krawitz3a
public double Type3b_Krawitz(int t){
        int p = -1, tot = 0;

        //this is another unique comment
        for (p = 0; p < t; p++)
                tot += p;

        if (t == 0)
                return (double)tot;
        else
                return (double)tot / t;
}
```

Figure 33. Type-3 Clones in Injected Code.



```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
```

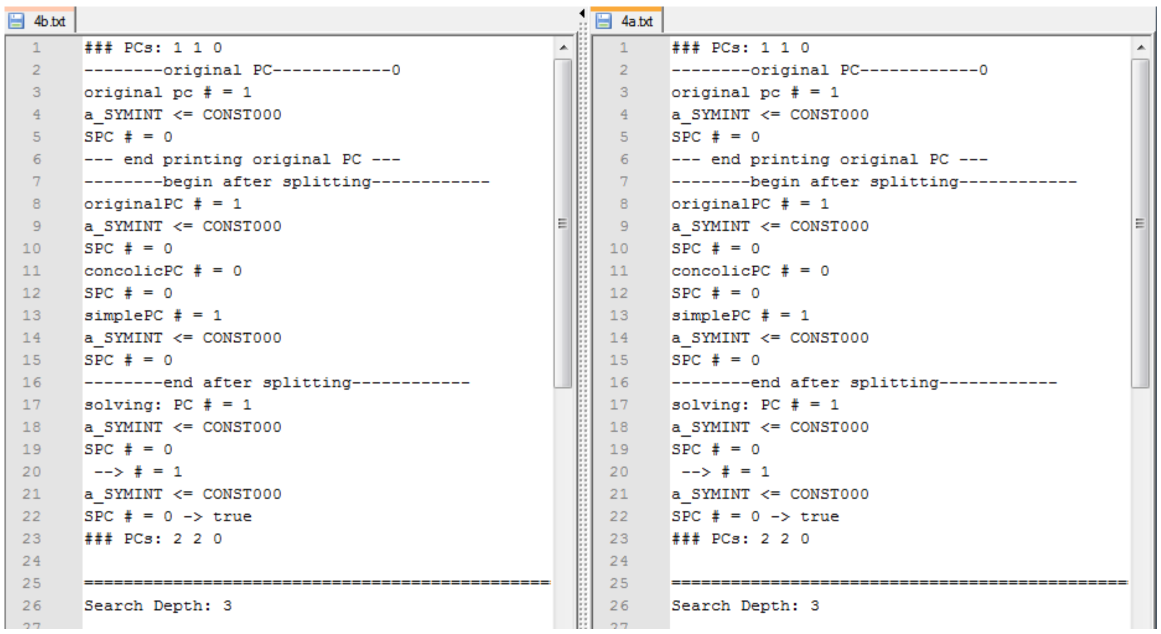
```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
```

Figure 34. Type-3 Concolic Output of Figure 33.

Type-3 clones have differences between methods that include added, removed or altered statements (Koschke, 2007). An example of a type-3 clone as defined by Krawitz (2012) is shown in Figure 33. The output generated by CCCD is shown in Figure 34. A *diff* between these two sets of concolic output states that they are identical, thus properly indicating a code clone candidate. Due to the extreme length of the concolic output, an abbreviated segment is shown in Figure 34. Complete results for all injected clones may be found in Appendix C.

```
//Type-4 Clones  - Krawitz4a
public double Type4a_Krawitz(int limit){
//to prevent stack overflow when large random values are input
        if (limit > 1000 || limit < 1)
                limit = 1;

        double[] d = new double[limit];
        double tot = 0;

        for (int n = 0; n < d.length; n++)
                d[n] = n * n * n;

        for (int n = 0; n < d.length; n++)
                tot += d[n];

        return tot;
}

// Type 4 Clone - Krawitz4b
public double Type4b_Krawitz(int limit){
        //to prevent stack overflow when large random values are input
        if (limit > 1000 || limit < 1)
                limit = 1000;

        return Type4b2_Krawitz("-", limit, 0, 0);
}
```

Figure 35. Type-4 Clones in Injected Code.


4b.txt

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_SYMINT <= CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT <= CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT <= CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT <= CONST000
SPC # = 0
 --> # = 1
a_SYMINT <= CONST000
SPC # = 0 -> true
### PCs: 2 2 0

===============================================
Search Depth: 3
```

4a.txt

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_SYMINT <= CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT <= CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT <= CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT <= CONST000
SPC # = 0
 --> # = 1
a_SYMINT <= CONST000
SPC # = 0 -> true
### PCs: 2 2 0

===============================================
Search Depth: 3
```

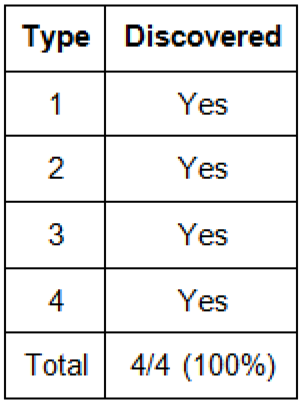
Figure 36. Type-4 Concolic Output of Figure 35

Type-4 clones are methods which produce the same result, but by using different syntax (Koschke, 2007). An example of a type-4 clone as defined by Krawitz (2012) is shown in Figure 35. The output from each of these methods is shown in Figure 36. A *diff* on these two sets of concolic output indicates that the two sets of output are identical, thus properly indicating a clone candidate. Due to the significant length of the concolic output, an abbreviated segment is shown in Figure 36. Complete results for all injected clones may be found in Appendix C.

Table 1. Roy Results

ID	Type	Discovered	Notes
1a	1	Yes	
1b	1	Yes	
1c	1	Yes	
2a	2	Yes	
2b	2	Yes	
2c	2	Yes	
2d	2	Yes	
3a	3	Yes	
3b	3	Yes	
3c	3	Partial	Modified JPF cannot process for technical reasons
3d	3	Yes	
3e	3	Yes	
4a	4	Yes	
4b	4	Yes	
4c	4	Yes	
4d	4	Yes	

Table 2. Krawitz Results.



Type	Discovered
1	Yes
2	Yes
3	Yes
4	Yes
Total	4/4 (100%)

Discovering all four clone types was very important because it demonstrated the robustness and effectiveness of CCCD. In order to further demonstrate its abilities, CCCD was run in a similar fashion against several other clones already defined by Krawitz (2012) and Roy (2009). A full listing of these clones is represented in Appendix A. As shown in Table 1, CCCD was able to discover clones defined by Roy (2009) very effectively. All types of clones were found. The only issues arose when CCCD attempted to analyze one of the clones as defined by Roy (2009), known as “3c” which is represented in Appendix A. JPF was unable to traverse all paths of this method for technical reasons. This is a limitation of JPF, not of the CCCD process. This constraint will be further discussed in Chapter 5 of this work. Even with this issue, CCCD was able to discover clones defined by Roy (2009) very proficiently. Table 2 displays CCCD’s ability to discover code clone candidates based upon the work by Krawitz (2012). CCCD was able to properly detect all of these clones.

Table 3. Existing Clones in Template Applications.

Application	Type-1 Clones	Type-2 Clones	Total
RES	1/1	2/2	3/3
JRand	3/3	1/1	4/4
DrJava	3/3	0	3/3
Jabref	4/4	0	4/4
JDraw	1/1	0	1/1
PS	1/1	0	1/1
TuxGuitar	1/1	0	1/1
Total	14/14	3/3	17/17

The next phase was to look for clones in the benchmark software. The examined open source applications were RES, JRand, DrJava, Jabref, JDraw, PS and TuxGuitar. Clones were pre-identified in the benchmark applications by using a manual process. Only type-1 and type-2 clones were discovered using this technique. This is not surprising since manually identifying type-3 and type-4 clone candidates is very difficult due to their semantic differences. CCCD was then ran against these applications to see if all of the manually identified clones could be discovered using this technique. CCCD was able to discover all 14 type-1 clones and the 3 identified type-2 clones in these applications. A complete set of these results may be seen in Table 3.

Table 4. Injected Clones in Template Applications.

Application	Type-1	Type-2	Type-3	Type-4	Total
RES	Yes	Yes	Yes	Yes	4/4
JRand	Yes	Yes	Yes	Yes	4/4
DrJava	Yes	Yes	Yes	Yes	4/4
Jabref	Yes	Yes	Yes	Yes	4/4
JDraw	Yes	Yes	Yes	Yes	4/4
PS	Yes	Yes	Yes	Yes	4/4
TuxGuitar	Yes	Yes	Yes	Yes	4/4
Total	7/7	7/7	7/7	7/7	28/28

In order to determine if CCCD was able to detect all four types of clones in these benchmark applications, clones identified by Krawitz (2012) and Roy (2009) were randomly injected into the source code and their locations were noted. A class containing all four types of clones was inserted into each application. The example class with all four clone types is represented in Appendix A. CCCD was then ran against these applications to see if all four types of clones could be identified in this sample class. For each application, CCCD was able to identify all of the inserted clones in this class.

The next step was to randomly insert clones defined by Krawitz (2012) and Roy (2009) into these applications. CCCD was then run against these applications to see which of these injected clones it would be able to discover. CCCD was able to identify all 28 clones injected into these applications. These results are represented in Table 4.

Summary

CCCD has been shown to be able to detect code clone candidates of all types using concolic analysis. CCCD was able to discover potential clones at the method level. Manual analysis was first used to find potential clones in several open source benchmark applications. CCCD was able to identify all of these manually identified clones. More clones of all types were taken from previous research and then injected into these applications. CCCD was able to discover all of these clones. CCCD had no trouble discovering all four types of clones, even the most difficult type-3 and type -4 clones.

Chapter 5

Conclusions

Concolic analysis has been demonstrated to be a highly effective code clone discovery mechanism capable of finding all types of clones. Concolic analysis was the primary mechanism for a developed clone detection process, CCCD. The semantics of the examined source code had no effect on CCCD's detection capabilities since concolic analysis only relies on the flow of the application and its possible paths. Things like naming conventions and comments which have been problematic for previous clone detection techniques have no bearing on CCCD (Roy et al., 2009). Discovering code clones is important in the field of software development (Bellon, Koschke, Antoniol, Krinke, & Merlo, 2007; Higo et al., 2007; Hummel et al., 2011). Clones increase the size and complexity of an application. This makes maintenance more complex and expensive. This increased size makes program comprehension more difficult (Geiger et al., 2006; Gode & Koschke, 2009).

CCCD worked by first performing concolic analysis on the source code of the targeted application on a method by method basis. This was done using a modified version of JPF. The output was then recorded and duplicate concolic output was an indication of a clone candidate.

Implications

This dissertation demonstrated the ability of concolic analysis to identify all four types of clones. Methods of ensuring proper path coverage using lazy instantiation for concolic analysis were also discussed. This is important because improper path coverage could lead to code clones being misidentified, either as false positives or false negatives. This dissertation also discussed proper methods of preparing the concolic output for comparison. Unneeded values from concolic analysis were removed. Improperly removing these values could also lead to inaccurate clone detection results.

Recommendations

CCCD was developed in order to prove the ability of concolic analysis to act as the basis for a candidate code clone discovery technique. While this tool was very effective in demonstrating these capabilities, it is by no means a complete application. Further work is needed to make clone detection into a more automated process. Additionally, it would be useful for the application to internally perform a *diff* and automatically create a report with the clone candidates discovered in the target application. Enhancements to the actual concolic generation process would also help in avoiding fatal errors when unsupported code is encountered. While this was not a significant problem in this work, this can foreseeably be an issue when this tool is applied on a much larger scale.

During the concolic analysis process, fatal exceptions would occur when JPF encountered an unsupported variable type. Some these unsupported variable types include short, byte and float. This is not a significant concern for several reasons. First of all, JPF is merely a concolic analysis component of CCCD. Future work on JPF or the inclusion of another concolic analysis tool into CCCD would likely solve this problem. Secondly, CCCD was still able to proficiently discover type-3 and even the tougher type-4 clones.

CCCD was successful at discovering clones on the method level. Future work may be done in order to allow concolic analysis to detect clones at a more granular level. This work would entail modifying JPF or the selected concolic analysis tool. These modifications were not done in this dissertation because the goal of this work was to merely demonstrate the feasibility of discovering code clones using concolic analysis.

Summary

Many software systems exist for extended periods of time. These applications will typically need to be updated in order to add new functionality and have bugs repaired (Kim et al., 2011). During these updates, functionality will often be duplicated in several areas of the application (Marcus & Maletic, 2001). This can occur for a variety of reasons. The first is that developers may not be aware that they are replicating this functionality. Applications are frequently very large, and developers often join and leave the project teams intermittently throughout its lifecycle. This makes it extremely difficult for developers to have a thorough understanding of the system. This lack of program comprehension may lead to developers unknowingly duplicating functionality throughout the application (Meneely et al., 2008). Developers may also knowingly repeat functionality in an application. In order to save time, they may copy and paste source code to several areas of the application, a process which is detrimental from a software engineering perspective (Pressman, 2010). These repeated segments often comprise a significant portion of a software project. One estimate is that between 5-23% of all code in software is redundant or exact copy and pastes of the source code (Baxter et al., 1998; Schulze et al., 2010).

Code clones are defined as multiple code fragments which produce similar results when given the same input (Fukushima et al., 2009). There are generally four types of accepted code clones (Roy et al., 2009). These range in complexity from the simpler type-1 to the more complex type-4 clones. There four defined levels of clones as described by Gold, Krinke, Harman, and Binkley (2010) are:

- Type-1: The code is syntactically identical except for white spaces, layout and comments.
- Type-2: Code is syntactically identical except for variations in identifiers, literals, types, and variations permitted under Type 1.
- Type-3: Code which is modified by adding, removing, or alteration statements, in addition to variations allowed under Type 2.
- Type-4: Code which uses different syntax, but produces the same result.

Clones are generally considered to be detrimental for a variety of reasons. First of all, they increase the maintenance costs of an application. This is because changes will need to be made and tested in numerous locations throughout the application (Juergens et al., 2009). Additionally, if changes are made inconsistently, this could lead to faults persisting in the application when changes to specific clones are overlooked (Deissenboeck et al., 2010). Additionally, tangled or scattered code will make it more difficult for developers to fully and properly understand the code base. This could lead to longer time being required for program comprehension (Kapser & Godfrey, 2008).

Several existing categories for clone detection techniques exist (Bruntink et al., 2005; Kim et al., 2011; Krawitz, 2012; Roy et al., 2009). These are:

- Text: Attempt to detect similar sequences by using minimal analysis.
- Lexical/Token: Apply lexical analysis to the source code and attempts to locate similar lines of code.
- Tree: Obtain a syntactical representation of the source code by using parsers.
- Metrics: Related to hashing algorithms. In this methodology, each fragment of a program, a number of various metrics are gathered regarding them. This information is subsequently used to find similar fragments.
- Graph: Obtains source code representation from a high level of abstraction.

- Program Dependency Graphs (PDGs) are comprised of information of a semantic manner. These include data such as control and data flow of the program.
- Functional: Performs black box testing on blocks of code. Clones produce identical outputs when provided identical inputs.
- Symbolic: Uses symbolic output of an application to discover similarities.

Many clone detection tools are only able to discover the simpler type-1 and type-2 clones. Far fewer works claim the ability to discover type-3 clones (Roy et al., 2009). Only two techniques claim to be able to discover the most complicated types of clones, type-4. These techniques use functional analysis and a memory comparison based technique. A primary drawback to the functional analysis process is that random data needs to be generated in order to discover clones. This can be a difficult and time consuming process (Krawitz, 2012). The memory comparison based technique suffers because it takes quite a long time to run and that it explores what is often an unreasonable large number of program states (Kim et al., 2011; Majumdar & Sen, 2007).

This dissertation introduced a new technique for discovering clone clones based on concolic analysis. CCCD is a tool which uses concolic analysis as the main component for detecting clones. CCCD first performs concolic analysis on the targeted source code using a modified version of JPF. Concolic analysis works by combining concrete and symbolic values in order to traverse all possible paths of an application (up to a given length) (Sen, 2007). Concolic analysis ultimately generates output indicating all possible paths an application may take (Majumdar & Sen, 2007; Sen et al., 2005).

Semantics, comments and infeasible paths are not taken into consideration. During this concolic analysis process, the modified version of JPF alters this generated output in order to remove unneeded information. This is accomplished through modifications made to the listener of this tool.

The final step of CCCD is a *diff* process conducted on this concolic output. As part of this proof of concept, an existing application known as Notepad++ carries out this phase. Duplicated output is an indication of a code clone candidate. This is because redundant output indicates that the paths or functionality of the application are identical. This identical functionality is a sign of a code clone candidate.

CCCD was verified using clones established by previous research (Krawitz, 2012; Roy et al., 2009). The first step was to confirm that CCCD was able to properly discover these previously identified clones on an individual basis, which it was successful in doing. The next phase was to verify that CCCD would be able to find clone candidates in existing programs. Several open source applications were selected and clones were manually identified in them. CCCD was then run against these programs and all of the pre-identified clones were successfully discovered by CCCD. All of these identified clones were of the simpler type-1 and type-2 categories. In order to check CCCD’s ability to discover the more complicated clones in existing applications, type-3 and type-4 clones were taken from previous research by Krawitz (2012) and Roy (2009). These clones were then injected into the selected open source applications. CCCD then examined the programs in order to check its ability to discover the clones. CCCD was able to discover all of the injected clones in these applications. During the development of CCCD, several questions had to be answered. These included the proper depth search, the use of lazy instantiation and how un-needed data could be removed from the concolic results.

This dissertation presented a new process for discovering code clones known as CCCD. Using concolic analysis, this technique found clone candidates based on the functionality of the application and not its syntactic nature. This means that things like naming conventions and comments in the source code had no effect on this clone detection process. CCCD was able to discover all four types of clones. The tool was verified using clones defined by several existing works and against manually identified existing clones in benchmark applications.

Appendix A

CCCD Validation Data for Type-1, Type-2, Type-3 and Type-4 Clones

```
import java.lang.Math;

public class Basic_Class1 {

	// Example of a dummy, non-clone function
	public void foo1(int a){
		if(a <3){
			while(a <3){
				a = a+1;
				System.out.println("while");
			}
		}
	}


	// Example of a dummy, non-clone function
	public int foo2(int a, int b)
	{
		if(a>b){
			b = a;
		}
		return a;
	}

	// Example of a dummy, non-clone function
	public int foo3(int a)
	{
		for (int i=0; i<a;i++){
			a = a+1;
		}
		return a;
	}

	// Example of a dummy, non-clone function
	public boolean foo4(int a){
		if (a>3){
			return true;
		}else{
			return false;
		}
	}
```

```
// Note: The following clones were taken from work by Krawitz(2012)

      //Type-1 Clones - Krawitz
      public double Type1a_Krawitz(int n)
      {
      int p = -1;
      int sum = 0;

      for (p = 0; p < n; p++)
      {
            sum += p;
      }

      if (n == 0) return sum;
      else return sum / n;
      }

      // Type 1 Clone - Krawitz
      public double Type1b_Krawitz(int n)
      {
      int p = -1;
      int sum = 0;

      //this is a comment that is not in any other method()
      for (p = 0; p < n; p++)
            sum += p;

      if (n == 0)
            return sum;
      else
            return sum / n;
      }



      //Type-2 Clones - Krawitz
      public double Type2a_Krawitz(int n)
      {
      int q = -1;
      double sum = 0;

      for (q = 0; q < n; q++)
      {
      sum += q;
      }

      if (n == 0) return sum;
      else return sum / n;
      }

      //Type-2 Clones - Krawitz
      public double Type2b_Krawitz(int t)
      {
      int p = -1;
      int tot = 0;

      //this is a comment that is not the same as any other comment
      for (p = 0; p < t; p++)
      tot += p;

      if (t == 0)
      return tot;
      else
      return tot / t;
      }


      // type 3 clone from Krawitz
      public double Type3a_Krawitz(int n)
      {
      int q = -1;
      double sum = 0;

      q = 0;
      while(q < n)
      {
      sum += q;
      q++;
      }

      if (n == 0) return sum;
      else return sum / n;
      }

      // type 3 clone - Krawitz
      public double Type3b_Krawitz(int t)
      {
      int p = -1, tot = 0;

      //this is another unique comment
      for (p = 0; p < t; p++)
            tot += p;

      if (t == 0)
            return (double)tot;
      else
            return (double)tot / t;
      }
```

```
      //Type-4 Clones  - Krawitz
      public double Type4a_Krawitz(int limit)
      {
      //to prevent stack overflow when large random values are input
      if (limit > 1000 || limit < 1)
            limit = 1;

      double[] d = new double[limit];
      double tot = 0;

      for (int n = 0; n < d.length; n++)
            d[n] = n * n * n;

      for (int n = 0; n < d.length; n++)
            tot += d[n];

      return tot;
      }

      // Type 4 Clone - Krawitz
      public double Type4b_Krawitz(int limit)
      {
      //to prevent stack overflow when large random values are input
      if (limit > 1000 || limit < 1)
            limit = 1000;

      return Type4b2_Krawitz("-", limit, 0, 0);
      }

       public double Type4b2_Krawitz(String s, int limit, double tot,
int n){

      if (limit > 1000 || limit < 1)//to prevent stack overflow
            limit = 1000;

      if (n < limit)
      tot = Type4b2_Krawitz("-", limit, tot + Math.pow(n, 3), ++n);
      return tot;
      }




// Note, these clones were taken from the work by Cordy(2008)

      // Original Code - Cordy
      void sumProdO(int n) {
            double sum=0.0; //C1
            double prod =1.0;
            for (int i=1; i<=n; i++)
            {
                  sum=sum + i;
                  prod = prod * i;
                  foo(sum, prod);
            }
      }

// Example 1A - Type 1 Clone - Cordy
      void sumProd1A(int n) {
            double sum=0.0; //C1
            double prod =1.0;
                  for (int i=1; i<=n; i++)
                  {
                        sum=sum + i;
                        prod = prod * i;
                        foo(sum, prod);
                  }
      }

      // Example 1B - Type 1 Clone - Cordy
      void sumProd1B(int n) {
            double sum=0.0; //C1
            double prod =1.0; //C
            for (int i=1; i<=n; i++)
            {
                  sum=sum + i;
                  prod = prod * i;
                  foo(sum, prod);
            }
      }

// Example 1C - Type 1 Clone - Cordy
      void sumProd1C(int n) {
            double sum=0.0; //C1
            double prod =1.0;
            for (int i=1; i<=n; i++) {
                  sum=sum + i;
                  prod = prod * i;
                  foo(sum, prod);
            }
      }


      // Example 2A - Type 2 Clone - Cordy
      void sumProd2A(int n){
            double s=0.0; //C1
            double p =1.0;
            for (int j=1; j<=n; j++)
            {
                  s=s + j;
                  p = p * j;
                  foo(s, p);
            }
      }
```

```
// Example 2B - Type 2 Clone - Cordy
void sumProd2B(int n){
      double s=0.0; //C1
      double p =1.0;
      for (int j=1; j<=n; j++)
      {
            s=s + j;
            p = p * j;
            foo(p, s);
            }
      }
```

```
// Example 2C - Type 2 Clone - Cordy
void sumProd2C(int n) {
      int sum=0; //C1
      int prod =1;
      for (int i=1; i<=n; i++)
      {
            sum=sum + i;
            prod = prod * i;
            foo(sum, prod);
            }
      }
```

```
// Example 2D - Type 2 Clone - Cordy
void sumProd2D(int n) {
      double sum=0.0; //C1
      double prod =1.0;
      for (int i=1; i<=n; i++)
      {
            sum=sum + (i*i);
            prod = prod*(i*i);
            foo(sum, prod);
            }
      }
```

```
// Example 3A - Type 3 Clone - Cordy
void sumProd3A(int n) {
      double sum=0.0; //C1
      double prod =1.0;
      for (int i=1; i<=n; i++)
      {
            sum=sum + i;
            prod = prod * i;
            foo(sum, prod, n);
      }
}
```

```
// Example 3B - Type 3 Clone - Cordy
void sumProd3B(int n) {
	double sum=0.0; //C1
	double prod =1.0;
	for (int i=1; i<=n; i++)
	{
		sum=sum + i;
		prod = prod * i;
		foo(prod);
		}
	}


	// Example 3C - Type 3 Clone - Cordy
void sumProd3C(int n) {
	double sum=0.0; //C1
	double prod =1.0;
	for (int i=1; i<=n; i++)
	{
		sum=sum + i;
		prod = prod * i;
		if ((n % 2) == 0) {
			foo(sum, prod);
		}
	}
}


	// Example 3D - Type 3 Clone - Cordy
void sumProd3D(int n) {
	double sum=0.0; //C1
	double prod =1.0;
	for (int i=1; i<=n; i++)
	{
		sum=sum + i;
		//line deleted
		foo(sum, prod);
		}
	}

// Example 3E - Type 3 Clone - Cordy
// For syntax purposes, the precise functionality was altered.
	public void sumProd3E(int n) {
	double sum=0.0; //C1
	double prod =1.0;
	for (int i=1; i<=n; i++)
	{
		if (i %2 == 0){
			sum+= i;
		}
		prod = prod * i;
		foo(sum, prod);
		}
}

// Example 4a - Type 4 Clone - Cordy
void sumProd4A(int n) {
      double prod =1.0;
      double sum=0.0; //C1
      for (int i=1; i<=n; i++)
      {
            sum=sum + i;
            prod = prod * i;
            foo(sum, prod);
      }
}


// Example 4B - Type 4 Clone - Cordy
void sumProd4B(int n) {
      double sum=0.0; //C1
      double prod =1.0;
      for (int i=1; i<=n; i++)
      {
            prod = prod * i;
            sum=sum + i;
            foo(sum, prod);
      }
}


// Example 4C - Type 4 Clone - Cordy
void sumProd4C(int n) {
      double sum=0.0; //C1
      double prod =1.0;
      for (int i=1; i<=n; i++)
      {
            sum=sum + i;
            foo(sum, prod);
            prod=prod * i;
      }
}


// Example 4D - Type 4 Clone - Cordy
void sumProd4D(int n) {
      double sum=0.0; //C1
      double prod =1.0;
      int i=0;
      while (i<=n)
      {
            sum=sum + i;
            prod = prod * i;
      }
}

// dummy methods to simply handle the test sum prod functions
```

```
private double foo(double sum)[ return sum +1.0; ]

private double foo(double sum, double prod, double temp)[
      return sum + prod + temp;
]

private double foo(double sum, double prod)[ return sum + prod +1; ]
```

Appendix B

Depth Search Examples

Depth Search Limit of 1

```
No path conditions for Run(0,java.lang.String@133,java.lang.String@135)
```

Depth Search Limit of 3

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
 --> # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0 -> false
### PCs: 2 1 1

====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
```

```
  call stack:
      at ProxyRun.depthTest(ProxyRun.java:82)
      at ProxyRun.Run(ProxyRun.java:33)
      at ProxyRun.main(ProxyRun.java:13)

--------original PC------------0
original pc # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
 --> # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0 -> true
### PCs: 3 2 1
bottom

====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at DrJava.toStringA(DrJava.java:21)
      at ProxyRun.Run(ProxyRun.java:61)
      at ProxyRun.main(ProxyRun.java:13)
```

Depth Search Limit of 5

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
```

```
a_1_SYMINT < a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0
 --> # = 1
a_1_SYMINT < a_1_SYMINT
SPC # = 0 -> false
### PCs: 2 1 1
--------original PC------------0
original pc # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
 --> # = 1
a_1_SYMINT >= a_1_SYMINT
SPC # = 0 -> true
### PCs: 3 2 1
bottom
--------original PC------------0
original pc # = 2
Length_0_ == CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
Length_0_ == CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ == CONST_0
SPC # = 0
--------end after splitting------------
solving: PC # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ == CONST_0
SPC # = 0
 --> # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ == CONST_0
SPC # = 0 -> false
### PCs: 4 2 2
--------original PC------------0
original pc # = 2
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0
SPC # = 0
--------end after splitting------------
solving: PC # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0
SPC # = 0
 --> # = 2
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0
SPC # = 0 -> true
### PCs: 5 3 2
--------original PC------------0
original pc # = 3
CONST_1 >= Length_0_ &&
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST_1 >= Length_0_ &&
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
```

```
SPC # = 0
simplePC # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 >= Length_0_
SPC # = 0
--------end after splitting------------
solving: PC # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 >= Length_0_
SPC # = 0
 --> # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 >= Length_0_
SPC # = 0 -> true
### PCs: 6 4 2
--------begin after splitting------------
originalPC # = 3
CONST_1 >= Length_0_[1] &&
Length_0_[1] != CONST_0 &&
a_1_SYMINT[-1000000] >= a_1_SYMINT[-1000000]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
a_1_SYMINT[-1000000] >= a_1_SYMINT[-1000000] &&
Length_0_[1] != CONST_0 &&
CONST_1 >= Length_0_[1]
SPC # = 0
--------end after splitting------------
solving: PC # = 3
a_1_SYMINT[-1000000] >= a_1_SYMINT[-1000000] &&
Length_0_[1] != CONST_0 &&
CONST_1 >= Length_0_[1]
SPC # = 0
 --> # = 3
a_1_SYMINT[-1000000] >= a_1_SYMINT[-1000000] &&
Length_0_[1] != CONST_0 &&
CONST_1 >= Length_0_[1]
SPC # = 0 -> true
MethodInfo[ProxyRun.main([Ljava/lang/String;)V]

====================================================== search constraint
Search Depth: 5

====================================================== snapshot
no live threads
--------original PC------------0
original pc # = 3
CONST_1 < Length_0_ &&
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST_1 < Length_0_ &&
Length_0_ != CONST_0 &&
a_1_SYMINT >= a_1_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 < Length_0_
SPC # = 0
--------end after splitting------------
solving: PC # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 < Length_0_
SPC # = 0
 --> # = 3
a_1_SYMINT >= a_1_SYMINT &&
Length_0_ != CONST_0 &&
CONST_1 < Length_0_
SPC # = 0 -> true
### PCs: 7 5 2

====================================================== search constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,priority=
5,lockCount=0,suspendCount=0
  call stack:
      at DrJava.toStringA(DrJava.java:28)
      at ProxyRun.Run(ProxyRun.java:61)
      at ProxyRun.main(ProxyRun.java:13)

PC # = 3
CONST_1 >= Length_0_[1] &&
Length_0_[1] != CONST_0 &&
a_1_SYMINT[-1000000] >= a_1_SYMINT[-1000000]
SPC # = 0
```

Appendix C

Extended Concolic Output

Krawitz 1a

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
--------original PC------------0
original pc # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 4 4 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1a_Krawitz(Basic_Class1.java:59)
      at Diss.Basic_Super.Run(Basic_Super.java:24)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
```

```
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 5 5 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1a_Krawitz(Basic_Class1.java:64)
      at Diss.Basic_Super.Run(Basic_Super.java:24)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
```

```
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 6 6 0
--------original PC------------0
original pc # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> false
### PCs: 7 6 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
```

```
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1a_Krawitz(Basic_Class1.java:64)
      at Diss.Basic_Super.Run(Basic_Super.java:24)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 8 7 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1a_Krawitz(Basic_Class1.java:65)
      at Diss.Basic_Super.Run(Basic_Super.java:24)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
```

```
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 9 8 1
--------original PC------------0
original pc # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> true
### PCs: 10 9 1
--------begin after splitting------------
originalPC # = 2
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0
concolicPC # = 0
SPC # = 0
```

```
simplePC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]
--------original PC------------0
original pc # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 11 10 1
--------original PC------------0
original pc # = 3
REAL == CONST &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
REAL == CONST &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
```

```
SPC # = 0
simplePC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == CONST
SPC # = 0
```

Krawitz 1b

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
```

```
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
--------original PC------------0
original pc # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 4 4 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1b_Krawitz(Basic_Class1.java:74)
      at Diss.Basic_Super.Run(Basic_Super.java:25)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 5 5 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1b_Krawitz(Basic_Class1.java:77)
      at Diss.Basic_Super.Run(Basic_Super.java:25)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
```

```
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 6 6 0
--------original PC------------0
original pc # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> false
### PCs: 7 6 1
```

```
====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1b_Krawitz(Basic_Class1.java:78)
      at Diss.Basic_Super.Run(Basic_Super.java:25)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 8 7 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type1b_Krawitz(Basic_Class1.java:80)
      at Diss.Basic_Super.Run(Basic_Super.java:25)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 9 8 1
--------original PC------------0
original pc # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> true
### PCs: 10 9 1
--------begin after splitting------------
originalPC # = 2
```

```
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]
--------original PC------------0
original pc # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 11 10 1
--------original PC------------0
original pc # = 3
REAL == CONST &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
```

```
REAL == CONST &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == CONST
SPC # = 0
```

Krawitz 2a

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0

====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type2a_Krawitz(Basic_Class1.java:95)
      at Diss.Basic_Super.Run(Basic_Super.java:27)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
```

<u>Krawitz 2b</u>

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
```

```
====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type2b_Krawitz(Basic_Class1.java:111)
      at Diss.Basic_Super.Run(Basic_Super.java:28)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
```

Krawitz 3a

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
```

```
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
--------original PC------------0
original pc # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
```

```
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 4 4 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3a_Krawitz(Basic_Class1.java:132)
      at Diss.Basic_Super.Run(Basic_Super.java:30)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 5 5 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3a_Krawitz(Basic_Class1.java:138)
      at Diss.Basic_Super.Run(Basic_Super.java:30)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 6 6 0
--------original PC------------0
original pc # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> false
### PCs: 7 6 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3a_Krawitz(Basic_Class1.java:138)
      at Diss.Basic_Super.Run(Basic_Super.java:30)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
```

```
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 8 7 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3a_Krawitz(Basic_Class1.java:139)
      at Diss.Basic_Super.Run(Basic_Super.java:30)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 9 8 1
--------original PC------------0
original pc # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
```

```
--------begin after splitting------------
originalPC # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> true
### PCs: 10 9 1
--------begin after splitting------------
originalPC # = 2
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]
--------original PC------------0
original pc # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
```

```
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 11 10 1
--------original PC------------0
original pc # = 3
REAL == a_SYMINT &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
REAL == a_SYMINT &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0
 --> # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0 -> true
### PCs: 12 11 1
--------begin after splitting------------
originalPC # = 3
REAL[-10000.0] == a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
CONST >= a_SYMINT[-10000]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
```

```
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
SPC # = 0
 --> # = 3
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]

======================================================= search
constraint
Search Depth: 5

======================================================= snapshot
no live threads
PC # = 2
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0

PC # = 3
REAL[-10000.0] == a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
CONST >= a_SYMINT[-10000]
SPC # = 0
```

Krawitz 3b

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
```

```
CONST < a_SYMINT
SPC # = 0
 --> # = 1
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 2 2 0
--------original PC------------0
original pc # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 3 3 0
--------original PC------------0
original pc # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
```

```
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0 -> true
### PCs: 4 4 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3b_Krawitz(Basic_Class1.java:148)
      at Diss.Basic_Super.Run(Basic_Super.java:31)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
CONST >= a_SYMINT &&
CONST < a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
```

```
### PCs: 5 5 0

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3b_Krawitz(Basic_Class1.java:151)
      at Diss.Basic_Super.Run(Basic_Super.java:31)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0
 --> # = 2
CONST < a_SYMINT &&
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 6 6 0
--------original PC------------0
original pc # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT == CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
```

```
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> false
### PCs: 7 6 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3b_Krawitz(Basic_Class1.java:152)
      at Diss.Basic_Super.Run(Basic_Super.java:31)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
a_SYMINT != CONST &&
CONST >= a_SYMINT &&
CONST < a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 3
```

```
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 3
CONST < a_SYMINT &&
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 8 7 1

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type3b_Krawitz(Basic_Class1.java:154)
      at Diss.Basic_Super.Run(Basic_Super.java:31)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
CONST >= a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 1
CONST >= a_SYMINT
SPC # = 0
 --> # = 1
CONST >= a_SYMINT
SPC # = 0 -> true
### PCs: 9 8 1
--------original PC------------0
original pc # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT == CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT == CONST
SPC # = 0 -> true
### PCs: 10 9 1
--------begin after splitting------------
originalPC # = 2
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT[0] &&
a_SYMINT[0] == CONST
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]
--------original PC------------0
original pc # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 2
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
```

```
SPC # = 0
--------end after splitting------------
solving: PC # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0
 --> # = 2
CONST >= a_SYMINT &&
a_SYMINT != CONST
SPC # = 0 -> true
### PCs: 11 10 1
--------original PC------------0
original pc # = 3
REAL == a_SYMINT &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 3
REAL == a_SYMINT &&
a_SYMINT != CONST &&
CONST >= a_SYMINT
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0
 --> # = 3
CONST >= a_SYMINT &&
a_SYMINT != CONST &&
REAL == a_SYMINT
SPC # = 0 -> true
### PCs: 12 11 1
--------begin after splitting------------
originalPC # = 3
REAL[-10000.0] == a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
CONST >= a_SYMINT[-10000]
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 3
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
```

```
SPC # = 0
--------end after splitting------------
solving: PC # = 3
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
SPC # = 0
 --> # = 3
CONST >= a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
REAL[-10000.0] == a_SYMINT[-10000]
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]

====================================================== search
constraint
Search Depth: 5

====================================================== snapshot
no live threads
PC # = 2
a_SYMINT[0] == CONST &&
CONST >= a_SYMINT[0]
SPC # = 0

PC # = 3
REAL[-10000.0] == a_SYMINT[-10000] &&
a_SYMINT[-10000] != CONST &&
CONST >= a_SYMINT[-10000]
SPC # = 0
```

Krawitz 4a

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_SYMINT <= CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT <= CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT <= CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT <= CONST000
SPC # = 0
 --> # = 1
```

```
a_SYMINT <= CONST000
SPC # = 0 -> true
### PCs: 2 2 0

======================================================= search
constraint
Search Depth: 3

======================================================= snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type4a_Krawitz(Basic_Class1.java:166)
      at Diss.Basic_Super.Run(Basic_Super.java:33)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
a_SYMINT > CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT > CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT > CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT > CONST000
SPC # = 0
 --> # = 1
a_SYMINT > CONST000
SPC # = 0 -> true
### PCs: 3 3 0
--------begin after splitting------------
originalPC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
 --> # = 1
a_SYMINT[1001] > CONST000
SPC # = 0 -> true
```

```
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]

====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
no live threads
PC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
```

Krawitz 4b

```
### PCs: 1 1 0
--------original PC------------0
original pc # = 1
a_SYMINT <= CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT <= CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT <= CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT <= CONST000
SPC # = 0
 --> # = 1
a_SYMINT <= CONST000
SPC # = 0 -> true
### PCs: 2 2 0

====================================================== search
constraint
Search Depth: 3

====================================================== snapshot
thread
index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,pr
iority=5,lockCount=0,suspendCount=0
  call stack:
      at Diss.Basic_Class1.Type4b_Krawitz(Basic_Class1.java:184)
      at Diss.Basic_Super.Run(Basic_Super.java:34)
      at Diss.Basic_Super.main(Basic_Super.java:11)

--------original PC------------0
original pc # = 1
```

```
a_SYMINT > CONST000
SPC # = 0
--- end printing original PC ---
--------begin after splitting------------
originalPC # = 1
a_SYMINT > CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT > CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT > CONST000
SPC # = 0
 --> # = 1
a_SYMINT > CONST000
SPC # = 0 -> true
### PCs: 3 3 0
--------begin after splitting------------
originalPC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
concolicPC # = 0
SPC # = 0
simplePC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
--------end after splitting------------
solving: PC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
 --> # = 1
a_SYMINT[1001] > CONST000
SPC # = 0 -> true
MethodInfo[Diss.Basic_Super.main([Ljava/lang/String;)V]

======================================================= search
constraint
Search Depth: 3

======================================================= snapshot
no live threads
PC # = 1
a_SYMINT[1001] > CONST000
SPC # = 0
```

References

Akito, M., & Shinichi, S. (2001). Capturing industrial experiences of software maintenance using product metrics.

Artzi, S., Dolby, J., Tip, F., & Pistoia, M. (2010). *Directed test generation for effective fault localization*. Paper presented at the Proceedings of the 19th international symposium on Software testing and analysis, Trento, Italy.

Baker, B. S. (1995). *On finding duplication and near-duplication in large software systems*. Paper presented at the Proceedings of the Second Working Conference on Reverse Engineering.

Bardin, S., & Herrmann, P. (2009). *Pruning the Search Space in Path-Based Test Generation*. Paper presented at the Proceedings of the 2009 International Conference on Software Testing Verification and Validation.

Basit, H. A., & Jarzabek, S. (2005). Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes, 30*(5), 156-165. doi: 10.1145/1095430.1081733

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). *Clone Detection Using Abstract Syntax Trees*. Paper presented at the Proceedings of the International Conference on Software Maintenance.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng., 33*(9), 577-591. doi: 10.1109/tse.2007.70725

Bond, M. D., Srivastava, V., McKinley, K. S., & Shmatikov, V. (2010). *Efficient, context-sensitive detection of real-world semantic attacks*. Paper presented at the Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Toronto, Canada.

Bruntink, M., Deursen, A. v., Engelen, R. v., & Tourwe, T. (2005). On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Trans. Softw. Eng., 31*(10), 804-818. doi: 10.1109/tse.2005.114

Bruntink, M., Deursen, A. v., & Tourwe, T. (2004). *An Initial Experiment in Reverse Engineering Aspects*. Paper presented at the Proceedings of the 11th Working Conference on Reverse Engineering.

Cadar, C., Godefroid, P., Khurshid, S., Pasreanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011). *Symbolic execution for software testing in practice: preliminary* *assessment*. Paper presented at the Proceeding of the 33rd international conference on Software engineering, Waikiki, Honolulu, HI, USA.

Chen, I.-X., Jaygarl, H., Yang, C.-Z., & Wu, P.-J. (2008). *Information retrieval on bug locations by learning co-located bug report clusters*. Paper presented at the Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, Singapore, Singapore.

Deissenboeck, F., Hummel, B., & Juergens, E. (2010). *Code clone detection in practice*. Paper presented at the Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, Cape Town, South Africa.

Duala-Ekoko, E., & Robillard, M. P. (2010). Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol., 20*(1), 1-31. doi: 10.1145/1767751.1767754

Ducasse, S., Rieger, M., & Demeyer, S. (1999). *A Language Independent Approach for Detecting Duplicated Code*. Paper presented at the Proceedings of the IEEE International Conference on Software Maintenance.

Fukushima, Y., Kula, R., Kawaguchi, S., Fushida, K., Nagura, M., & Iida, H. (2009). *Code Clone Graph Metrics for Detecting Diffused Code Clones*. Paper presented at the Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference.

Geiger, R., Fluri, B., Gall, H., & Pinzger, M. (2006). Relation of Code Clones and Change Couplings. In L. Baresi & R. Heckel (Eds.), *Fundamental Approaches to Software Engineering* (Vol. 3922, pp. 411-425): Springer Berlin / Heidelberg.

Gode, N., & Koschke, R. (2009). *Incremental Clone Detection*. Paper presented at the Proceedings of the 2009 European Conference on Software Maintenance and Reengineering.

Gold, N., Krinke, J., Harman, M., & Binkley, D. (2010). *Issues in clone classification for dataflow languages*. Paper presented at the Proceedings of the 4th International Workshop on Software Clones, Cape Town, South Africa.

Higo, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2007). Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol., 49*(9-10), 985-998. doi: 10.1016/j.infsof.2006.10.005

Hummel, B., Juergens, E., & Steidl, D. (2011). *Index-based model clone detection*. Paper presented at the Proceedings of the 5th International Workshop on Software Clones, Waikiki, Honolulu, HI, USA.

Ihantola, P. (2006). *Test data generation for programming exercises with symbolic execution in Java PathFinder*. Paper presented at the Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006, Uppsala, Sweden.

Jarzabek, S., & Xue, Y. (2010). *Are clones harmful for maintenance?* Paper presented at the Proceedings of the 4th International Workshop on Software Clones, Cape Town, South Africa.

Johnson, J. H. (1993). *Identifying redundancy in source code using fingerprints*. Paper presented at the Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1, Toronto, Ontario, Canada.

Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). *Do code clones matter?* Paper presented at the Proceedings of the 31st International Conference on Software Engineering.

Kalibera, T., Parizek, P., Malohlava, M., & Schoeberl, M. (2010). *Exhaustive testing of safety critical Java*. Paper presented at the Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, Prague, Czech Republic.

Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on, 28*(7), 654-670.

Kapser, C. J., & Godfrey, M. W. (2008). "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Softw. Engg., 13*(6), 645-692. doi: 10.1007/s10664-008-9076-6

Khurshid, S., Pasareanu, C. S., & Visser, W. (2003). *Generalized symbolic execution for model checking and testing*. Paper presented at the Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems, Warsaw, Poland.

Kim, H., Jung, Y., Kim, S., & Yi, K. (2011). *MeCC: memory comparison-based clone detector*. Paper presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA.

Koschke, R. (2007). Survey of Research on Software Clones. *Seminar on Deplication, Redundancy, and Similarity in Software*.

Krawitz, R. M. (2012). *Code Clone Discovery Based on Functional Behavior*

Lague, B., Proulx, D., Mayrand, J., Merlo, E. M., & Hudepohl, J. (1997). *Assessing the Benefits of Incorporating Function Clone Detection in a Development Process*. Paper presented at the Proceedings of the International Conference on Software Maintenance.

Lakhotia, K., Harman, M., & McMinn, P. (2008). *Handling dynamic data structures in search based testing*. Paper presented at the Proceedings of the 10th annual conference on Genetic and evolutionary computation, Atlanta, GA, USA.

Li, Z., & Garcia-Luna-Aceves, J. J. (2007). Finding multi-constrained feasible paths by using depth-first search. *Wirel. Netw., 13*(3), 323-334. doi: 10.1007/s11276-006-7528-8

Maisikeli, S. G., & Mitropoulos, F. J. (2010, 3-5 Oct. 2010). *Aspect mining using Self-Organizing Maps with method level dynamic software metrics as input vectors.* Paper presented at the Software Technology and Engineering (ICSTE), 2010 2nd International Conference on.

Majumdar, R., & Sen, K. (2007). *Hybrid Concolic Testing*. Paper presented at the Proceedings of the 29th international conference on Software Engineering.

Marcus, A., & Maletic, J. I. (2001). *Identification of High-Level Concept Clones in Source Code*. Paper presented at the Proceedings of the 16th IEEE international conference on Automated software engineering.

Mayrand, J., Leblanc, C., & Merlo, E. M. (1996, 4-8 Nov 1996). *Experiment on the automatic detection of function clones in a software system using metrics.* Paper presented at the Software Maintenance 1996, Proceedings., International Conference on.

Meneely, A., Williams, L., Snipes, W., & Osborne, J. (2008). *Predicting failures with developer networks and social network analysis*. Paper presented at the Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia.

Menon, V., Pingali, K., & Mateev, N. (2003). Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst., 25*(6), 776-813. doi: 10.1145/945885.945888

Monden, A., Nakae, D., Kamiya, T., Sato, S.-i., & Matsumoto, K.-i. (2002). *Software Quality Analysis by Code Clones in Industrial Legacy Software*. Paper presented at the Proceedings of the 8th International Symposium on Software Metrics.

Pasareanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., & Pape, M. (2008). *Combining unit-level symbolic execution and system-level concrete execution for testing nasa software*. Paper presented at the Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA.

Person, S., Dwyer, M. B., Elbaum, S., & Pasareanu, C. S. (2008). *Differential symbolic execution*. Paper presented at the Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia.

Person, S., Yang, G., Rungta, N., & Khurshid, S. (2011). Directed incremental symbolic execution. *SIGPLAN Not., 46*(6), 504-515. doi: 10.1145/1993316.1993558

Pressman, R. (2010). *Software Engineering A Practitioner's Approach* (7th ed.). Boston: McGraw Hill Higher Education.

Roy, C. K., & Cordy, J. R. (2008). *Scenario-Based Comparison of Clone Detection Techniques*. Paper presented at the Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension.

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program., 74*(7), 470-495. doi: 10.1016/j.scico.2009.02.007

Schulze, S., Apel, S., & Kastner, C. (2010). *Code clones in feature-oriented software product lines*. Paper presented at the Proceedings of the ninth international conference on Generative programming and component engineering, Eindhoven, The Netherlands.

Seaman, C. B. (2008). Software Maintenance: Concepts and Practice Authored by Penny Grubb and Armstrong A. Takang World Scientific, New Jersey. Copyright ; 2003; 349 pages ISBN 981-238-426-X (paperback) *J. Softw. Maint. Evol., 20*(6), 463-466. doi: 10.1002/smr.v20:6

Sen, K. (2007). *Concolic testing*. Paper presented at the Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA.

Sen, K., Marinov, D., & Agha, G. (2005). *CUTE: a concolic unit testing engine for C*. Paper presented at the Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal.

Shukla, R., & Misra, A. K. (2008). *Estimating software maintenance effort: a neural network approach*. Paper presented at the Proceedings of the 1st India software engineering conference, Hyderabad, India.

Sibeyn, J. F., Abello, J., & Meyer, U. (2002). *Heuristics for semi-external depth first search on directed graphs*. Paper presented at the Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, Winnipeg, Manitoba, Canada.

Singh, Y., Bhatia, P. K., & Sangwan, O. (2009). Predicting software maintenance using fuzzy model. *SIGSOFT Softw. Eng. Notes, 34*(4), 1-6. doi: 10.1145/1543405.1543425

Singh, Y., & Goel, B. (2007). A step towards software preventive maintenance. *SIGSOFT Softw. Eng. Notes, 32*(4), 10. doi: 10.1145/1281421.1281432

Takaki, M., Cavalcanti, D., Gheyi, R., Iyoda, J., D'Amorim, M., Prud\, R. B., . . . ncio. (2010). Randomized constraint solvers: a comparative study. *Innov. Syst. Softw. Eng., 6*(3), 243-253. doi: 10.1007/s11334-010-0124-1

Tatti, N., & Cule, B. (2011). *Mining closed episodes with simultaneous events*. Paper presented at the Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, San Diego, California, USA.

Ueda, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2002). *Gemini: Maintenance Support Environment Based on Code Clone Analysis*. Paper presented at the Proceedings of the 8th International Symposium on Software Metrics.

Visser, W., Pasareanu, C. S., & Khurshid, S. (2004). *Test input generation with java PathFinder*. Paper presented at the Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA.

Wahler, V., Seipel, D., Gudenberg, J. W. v., & Fischer, G. (2004). *Clone Detection in Source Code by Frequent Itemset Techniques*. Paper presented at the Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop.

Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., & Su, Z. (2008). *Dynamic test input generation for web applications*. Paper presented at the Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA.

Wettel, R., & Marinescu, R. (2005). *Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments*. Paper presented at the Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.

Yang, D., & Powers, D. M. W. (2005). *Measuring semantic similarity in the taxonomy of WordNet*. Paper presented at the Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38, Newcastle, Australia.

Yang, W. (1991). Identifying syntactic differences between two programs. *Softw. Pract. Exper., 21*(7), 739-755. doi: 10.1002/spe.4380210706