Exploring Equivalence and Differences in Software Methods

by

Tarcisio Teixeira

B.Sc., École Polytechnique, 2021

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

September 2023

© Tarcisio Teixeira, 2023
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Exploring Equivalence and Differences in Software Methods**

submitted by Tarcisio Teixeira in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examining Committee:**

Reid Holmes, Professor, Computer Science, UBC  
*Supervisor*

Elisa Baniassad, Professor of Teaching, Computer Science, UBC  
*Supervisory Committee Member*
Abstract

This thesis presents a novel technique for the comprehensive analysis and characterization of differences between two versions of a method. Building upon the foundations of Differential Symbolic Execution [13], our approach extends the concept of equivalence to effectively prove method functional equivalence or, when not equivalent, provide input constraints that narrow down the input space leading to divergent behaviors.

To evaluate the effectiveness of our approach, we developed a prototype tool and conducted an extensive study using the IntroClass benchmark for Java. Our research sought to answer two critical research questions: 1) the effectiveness of our approach in classifying functionally equivalent programs compared to a standard partition-effects equivalence technique, and 2) the robustness of our approach in restricting the input space to pinpoint behavioral differences.

The results of our evaluation demonstrate the significant potential of our approach in specifying the differences between method versions by characterizing the partition of the input space where those differences unfold. While our technique can establish program equivalence, we acknowledge that further evaluation and real-world application are needed to fully validate its practical competency.

This thesis contributes a promising methodology for analyzing and characterizing method differences, offering a valuable tool for software engineers and researchers in the field of program analysis and verification. Future research and real-world applications could provide additional insights into the practical benefits and limitations of this approach.
Lay Summary

This thesis introduces an innovative method for understanding the differences between two versions of a computer program. Imagine you have two similar computer programs, but you want to know precisely how they behave differently. Our approach, based on Differential Symbolic Execution, provides a solution. It can improve the range of programs for which we can prove two versions are functionally equivalent or, if they’re not, it can pinpoint the specific inputs that make them behave differently.

We developed a practical tool based on this approach and conducted a study aimed to answer two important questions: How well does our method identify equivalent programs, and how effectively does it narrow down the inputs causing differences?

The results are promising, showing that our approach is quite effective at specifying differences between program versions. However, more real-world testing is needed to confirm its practicality fully. In summary, this research offers a valuable tool for understanding and verifying program differences, with the potential to benefit software developers and researchers in program analysis and verification.
Preface

The work presented in this thesis was conducted in the Software Practices Laboratory at the University of British Columbia, Vancouver campus.

Tarcisio Teixeira was the lead investigator, responsible for the concept formulation, tool design and implementation, result analysis, and manuscript composition. Professor Reid Holmes was the supervisory author and was involved throughout all the steps of this project.
# Table of Contents

Abstract ................................................................. iii

Lay Summary ............................................................. iv

Preface ............................................................... v

Table of Contents .................................................... vi

List of Tables ......................................................... viii

List of Figures ......................................................... x

Acknowledgments ..................................................... xii

1 Introduction .......................................................... 1

2 Background and Related Work .................................... 4
   2.1 Symbolic Execution ...................................... 5
   2.2 DSE ......................................................... 6

3 Motivating Example ................................................ 10
   3.1 Interpretation Problem .................................. 10
   3.2 Elimination Problem ..................................... 12

4 Approach ............................................................ 14
   4.1 Interpretation .......................................... 14
   4.2 Elimination ........................................... 17
**List of Tables**

<table>
<thead>
<tr>
<th>Table 2.1</th>
<th>Table showing the semantic difference between versions 1 and 2 of method foo.</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 6.1</td>
<td>Results for the abstraction experiment. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. E1 processing method achieving the best results supporting the strength of our approach.</td>
<td>36</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Oracle Experiment results restricted to the Median kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % for E supporting the strength in restricting the input space.</td>
<td>37</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Oracle Experiment results restricted to the Smallest kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % for E supporting the strength in restricting the input space.</td>
<td>38</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>Versioning experiment results restricted to the Median kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % supporting it strength in restricting the input space.</td>
<td>39</td>
</tr>
</tbody>
</table>
Table 6.5  Versioning experiment results restricted to the Smallest kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % supporting it strength in restricting the input space.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Implementations of the method \texttt{foo}. In the left hand side version 1 and in the right hand side version 2. Difference occurs in line 7, in one version \texttt{glob} is assigned 0 and in the other 1. In gray is the common code block to be abstracted during analysis.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Version 3 of \texttt{foo} introduced in Figure 2.1. In gray is the common code block to be abstracted during analysis.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Version 4 of \texttt{foo} introduced in Figure 2.1. In gray is the common code block to be abstracted during analysis.</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Fictious method \texttt{untp1} whose summary is used in the interpretation algorithm.</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementations of \texttt{f}. Version 1 on the left hand side and Version 2 on the right hand side. In gray is a common code block to be abstracted during analysis.</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustration of modifications made in order to allow uninterpreted function applications as expressions in symbc JPF. We have the application of two uninterpreted functions \texttt{UNTP.C.1} and \texttt{UNTP.I.1_STATE}.</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Code snippet before any modifications are made to serve as input for our tool. In gray is the code block to be abstracted.</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Modified version of code snippet in Figure 5.2 to serve as input for our tool. In gray is the replacement for the abstracted code.</td>
<td>30</td>
</tr>
</tbody>
</table>
Figure B.1  Code snippet automatically generated from the benchmark code of Figure B.2. ................................. 58
Figure B.2  Original code example from IntroClass benchmark for Java. ................................. 59
Acknowledgments

I would like to take this opportunity to express my sincere gratitude to the individuals and groups who have been instrumental in the successful completion of this thesis.

I would like to express my gratitude to my supervisor, Professor Reid Holmes, whose guidance, encouragement, and expertise have been instrumental in the successful completion of this research.

I am immensely grateful to my fellow laboratory mates, whose camaraderie made the research journey not just productive, but also filled with fun and enriching experiences. Special thanks go to Yanze Li for the insightful late-night discussions in the lab, your perspectives and thought-provoking insights have played a pivotal role in shaping the ideas presented in this thesis.

To my family, I owe an immeasurable debt of gratitude. Your unflagging support, encouragement, and belief in my abilities have been the driving force behind my academic endeavors. Your patience and understanding during the challenges of this journey have been my anchor, and I am profoundly thankful for your unwavering presence in my life.

This research would not have been possible without the collective support and inspiration of many. To all those who have played a part, whether big or small, in this academic voyage, I extend my heartfelt thanks.
Chapter 1

Introduction

Software is evolving, each day companies make thousands of commits. The way developers evolve a piece of software is usually by making changes to its source code. After a change developers focus on two primary questions: “Did I successfully implement the intended change?” and “Did I break anything else?” [7]. In order to solve these questions one needs to reason about the program behavior of the new version compared with the previous one. Reasoning about behavior is hard especially for humans because it requires the developer to reason about the dynamic nature of the program [5].

Program behavior can be defined in several ways. For this project we consider two methods having the same behavior if, at the end of their execution, the output is the same and all the still-existent variables have the same value. In the case of pointers, we would compare the objects they point to instead of their memory values.

In practice, developers mostly use test suites for verifying behavioral differences. To evaluate whether the change was successfully implemented, they look at their newly added test cases. To check whether they broke anything they look at regression tests. Therefore, in most cases, the confidence a developer has concerning the answer to these two questions is module the tests suites.

Testing software is a hard task in general, throughout the years, several tools and techniques have been developed aiming at the automation of testing [3, 10, 15]. Furthermore, there has been a lot of work in the field of differential testing [11,
that is, the development of tools to help developers answer the two questions stated above using test suites. In these cases, the idea is to use symbolic execution and constraint solving to generate test inputs that explore the difference between versions.

Another slightly different research direction is differential program analysis which aims to prove semantic equivalence or to expose the behavioral difference between two versions of a piece of software. Work in this field tends to use static analysis and symbolic execution to produce code summaries capturing some sort of program behavior [1, 8, 13]. The vast majority of approaches to analyzing version semantic differences rely on symbolic execution. Although symbolic execution is effective in characterizing program behavior, it has one major problem: scalability. The large number of execution paths generated in a symbolic analysis execution makes scaling symbolic execution a challenge [14]. Although this problem is already eased by the nature of differentially analyzing code, it still impacts the complexity of the analysis.

In this thesis, we propose a technique to analyze two versions of a method and characterize their differences. Our approach extends Differential Symbolic Execution (DSE) introducing a new notion of equivalence. Our notion of equivalence is coarser-grained than the notion used in DSE and finer-grained than functional equivalence itself. Thus, augmenting the range of programs in which one is capable of proving method functional equivalence. Our approach relies on symbolic execution to build method summaries and a new algorithm for processing the summaries in order to prove equivalence. In the case where methods are not equivalent our approach provides an input constraint restricting the total input space to a partition that can produce different behavior between versions.

We build a prototype tool implementing our approach. Then, we conducted a study in order to evaluate its ability to fully characterize the difference between two versions of a method. We run our tool on the IntroClass benchmark for Java [6]. We assess the effectiveness of our approach in contrast to using the standard notion of equivalence defined in Differential Symbolic Execution. The thesis of this work is that our approach surpasses partition-effects equivalence based approaches in characterizing method behavior differences by 1) augmenting the range of programs for which we can prove two versions are functionally equivalent and 2) pro-
viding a more exact partition of the input space leading to behavioral divergence. This leads to the following two key research questions:

**RQ1.** How effective is our approach in classifying two functionally equivalent programs compared with the partition-effects equivalence technique?

**RQ2.** How robust is our approach in restricting the input space capturing only the inputs that expose the difference between versions?

Our evaluation supports the high potential our approach has to fully specify the difference between versions of a method. The study shows a significant improvement in fully specifying the partition of the input space that produces behavioral difference. Although, it can prove program equivalence our evaluation was not sufficient to show its competency in practice.
Chapter 2

Background and Related Work

There is a large variety of projects in the field of aiding developers succeed in evolving software. There is extensive research on automated testing and software verification that enhance developers confidence in about correctness and reliability. Our approach is focused in software comprehension and proving equivalence between methods. In this chapter we discuss some of the related work and how they differ from our approach. We selected previous projects that share a similar goal, focusing in formally capturing the difference among two versions of software.

Semantic Diff [8]. A tool that takes two versions of a procedure and generates a report summarizing the semantic differences between them. This tool expresses its results in terms of the observable input-output behaviour of the procedure, rather than its syntactic structure. The output of this tool is a list of pairs input-output where the input can somehow affect the value of the output. The tool proposed in this thesis differs by generating the constraints where a difference can be observed, not only a differential dependence relation between input-output.

Conditional Equivalence [1]. In this work, it is proposed the notion of conditional (partial) equivalence, that is checking if two versions are equivalent for a subset of the input space. In contrast with this thesis we are interested in analyzing the differences for the whole input space. The whole input space is in general undecidable but the use of uninterpreted functions may allow one to decide equivalence
DiSE [14]. This work presents a technique for improving symbolic execution. The general idea consists in pruning paths whose sequence of affected nodes have been covered. DiSE is a complementary technique to other reduction or bounding techniques developed to improve symbolic execution. Our approach differs from DiSE since we use standard Symbolic Execution to generate path constraints and then we process them using our novel algorithm.

DSE [13]. This work presents the concept of differential symbolic execution providing a starting point for this project. This project differs from DSE by improving the algorithm for differentiating summaries in two ways. First, it does not require partition-effect equivalence, that is, instead of eliminating two equivalent paths at each iteration, it tries to eliminate a contained paths from both versions. Second, it furthers lazily look at the inserted uninterpreted functions if needed.

We begin by briefly explaining some of the core concepts used in this project. In this chapter, we first give a short description of Symbolic Execution which is used in our approach to construct method summaries to be further analyzed. Then, we deepen our explanation of Differential Symbolic Execution (DSE) [13].

2.1 Symbolic Execution

Symbolic execution is a program analysis technique that aims to explore the behavior of a program by tracking the symbolic values of variables rather than concrete values [9]. In this technique, algebraic symbols are used as program input instead of actual values which are manipulated based on the program’s instructions and conditions. The key idea is to explore multiple execution paths by tracking how these symbols change and by recording the conditions under which certain paths are taken. The program output is an expression over these concrete values, actual values, and a specified set of operators.

A state of a symbolic execution can be represented by a program location identifier, a mapping from variables to their symbolic expressions, and a path condition. The mapping stores the current symbolic value each variable has while the
path condition is a constraint on the input that leads program execution to that point. During the execution, path conditions are assessed to determine if they can be satisfied. If a path condition proves to be unsatisfiable, symbolic execution ceases exploration of that particular path and backtracks. In programs with loops or recursions, the exploration could be possibly infinite, thus, there is usually a user-specified bound for the depth of the execution.

2.2 DSE

In DSE, they use Symbolic Execution to summarize both versions of a method and then they check whether the summaries are equivalent. If they are not equivalent they display the behavioral difference. In DSE, the methods are reduced to a set of partition-effect pairs as defined below.

**Definition 1 (Partition-Effects Pair).** A partition-effects pair, \((i, e)\), consists of: an input constraint, \(i\), which is a conjunction of relational expressions defined over constants and symbolic variables, and an effects constraint, \(e\), which is a conjunction of expressions that equate written locations to expressions defined over constants and symbolic variables.

The intuition behind partition-effects is to characterize the behavior of a method as a pair where the first element describe the partition of the input space and the second element the effects this partition causes. In this definition the effects are essentially a constraint indicating the final value of each variable still alive after the method execution. Following the definition of partition effects we can define symbolic summaries as below.

**Definition 2 (Symbolic Summary).** A symbolic summary, for a method \(m\), is a set of partition-effects pairs \(m_{\text{sum}} = \{(i_1, e_1), (i_2, e_2), \ldots, (i_k, e_k)\}\) where the input constraints are disjoint, i.e., \(\forall 1 \leq j \leq k \forall 1 \leq j' \leq k \land j \neq j' : i_j \land i_{j'}\) is unsatisfiable.

From this definition, it is important to highlight that the input constraints are disjoint, that is for a given input there is at most one element in the set corresponding to executing the method under analysis on it. Ideally, a Symbolic Summary
would account for the entire input space. However, in practice, these summaries are generated using symbolic execution and due to the halting problem we end up with incomplete summaries [16]. In order to overcome this problem they define abstract summaries.

**Definition 3 (Abstract Summary)**. An abstract summary, for a method \( m(\vec{f}) \) with parameters \( \vec{f} \) is a singleton set of partition-effects pairs

\[
m_{\text{abs}} = \{(i(\vec{f}), \bigwedge I \in \text{Write}(m) : l == e_l(\vec{f}))\}
\]

where \( \text{Write}(m) \) is the set of variable that the method \( m \) writes to, \( i : \vec{f} \rightarrow \{\text{true}, \text{false}\} \), and \( e_l : \vec{f} \rightarrow \text{typeof}(l) \) are uninterpreted functions defined over vectors of parameter values.

The intuition is to use uninterpreted functions to complete the summaries, the uninterpreted function can trivially account for a part of the input space uncovered. Moreover, although it may not provide a reader with some concrete idea about the input or the effects it allows reasoning and manipulation using function theory on SMT solvers [2].

Additionally one can use uninterpreted functions to abstract away the behavior of a piece of code. In DSE they combine both kinds of summaries to generate a method summary [13]. Those method summaries are then analyzed generating a table with the behavioral difference between different versions of a method.

Informally, we say that two versions of a method \( m \) are functionally equivalent if for all inputs the effects caused on both methods are the same, that is, the return values are the same and all variables written within the scope of \( m \) that are still alive after its execution are assigned to the same value. In terms of method summaries, we have the following definition.

**Definition 4 (Functional Equivalence)**. Two methods \( m, m' \) with respective symbolic summaries \( s \) and \( s' \) are functionally equivalent if and only if

\[
\left( \bigvee_{(i,e) \in s} i \land e \right) \iff \left( \bigvee_{(i',e') \in s'} i' \land e' \right)
\]

Deciding if two methods are functionally equivalent by purely checking the
Boolean equivalence above is in most cases not feasible due to exponential complexity on SAT solvers [4]. In DSE, instead of looking for straight into functional equivalence, they check whether two methods are partition-effects equivalent.

**Definition 5 (Partition-effects Equivalence).** Two methods \( m, m' \) with respective symbolic summaries \( s \) and \( s' \) are partition-effects equivalent if and only if

\[
\forall (i, e) \in s \; \exists (i', e') \in s' \; \text{s.t.} \; (i \land e) \iff (i' \land e')
\]

\[
\forall (i', e') \in s' \; \exists (i, e) \in s \; \text{s.t.} \; (i' \land e') \iff (i \land e)
\]

This notion of equivalence necessitates an isomorphism between method summaries, ensuring that every corresponding pair of elements holds logical equivalence. One can easily check that partition-effect equivalence implies functional equivalence. Therefore, the strategy in DSE is to check partition effect equivalence.

Consider the two versions of the method \( \text{foo} \) in Figure 2.1. The two versions have a common chunk of code highlighted in gray in Figure 2.1. Using the approach described in DSE we can abstract the common code and produce the following summaries, \( \text{foo1} \) and \( \text{foo2} \):

```java
public static int foo(int var) {
    if (var > 0) {
        if (glob > 0) var = var*var;
        else var = var*(var+1);
        glob = var;
    } else {
        glob = 0;
    }
    return var;
}
```

```java
public static int foo(int var) {
    if (var > 0) {
        if (glob > 0) var = var*var;
        else var = var*(var+1);
        glob = var;
    } else {
        glob = 1;
    }
    return var;
}
```

**Figure 2.1:** Implementations of the method \( \text{foo} \). In the left hand side version 1 and in the right hand side version 2. Difference occurs in line 7, in one version \( \text{glob} \) is assigned 0 and in the other 1. In gray is the common code block to be abstracted during analysis.
\( f_{oo1} = \{(V > 0 \land \text{UNT}P_1(V, G), \text{glob} = \text{glob}_1(V, G) \land \text{RETURN} = \text{var}_1(V, G)), \\
(\neg(V > 0), \text{glob} = 0 \land \text{RETURN} = V)\} \)

\( f_{oo2} = \{(V > 0 \land \text{UNT}P_1(V, G), \text{glob} = \text{glob}_1(V, G) \land \text{RETURN} = \text{var}_1(V, G)), \\
(\neg(V > 0), \text{glob} = 1 \land \text{RETURN} = V)\} \)

where \( G \) represents the symbolic value of the external variable \( \text{glob} \) when the method is invoked and \( V \) is the symbolic value of the input variable \( \text{var} \). The \( \text{UNT}P_1 \) function represents the path condition within the abstracted block and the functions \( x_1(V, G) \) represent the resultant values for variable \( x \) when the block completes execution. The idea here is to associate the common block with the following abstract summary. In this thesis, we refer to uninterpreted functions in the input constraint (e.g. \( \text{UNT}P_1 \)) of a partition-effect pair as \textit{uninterpreted constraint functions} and in the effects (e.g. \( \text{glob}_1, \text{var}_1 \)) as \textit{uninterpreted value functions}.

\[ s = \{(\text{UNT}P_1(V, G), \text{glob} = \text{glob}_1(V, G) \land \text{var} = \text{var}_1(V, G))\} \]

where \( V \) and \( G \) are respectively the symbolic values of variables \( \text{var} \) and \( \text{glob} \) before the execution of the block.

Analyzing the method summaries we see that the first element of each summary are logically equivalent, this way, the difference occurs only on the second one and we can derive from that Table 2.1

<table>
<thead>
<tr>
<th>On Input</th>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{var} \leq 0 )</td>
<td>( \text{glob} = 0 )</td>
<td>( \text{glob} = 1 )</td>
</tr>
</tbody>
</table>

**Table 2.1**: Table showing the semantic difference between versions 1 and 2 of method \( f_{oo} \).

Therefore providing the developer with information about the behavioral change which is useful for one to judge whether the differences were indeed intended.
Chapter 3

Motivating Example

In this chapter, we explore two drawbacks of the DSE approach when analyzing versions of code. The disadvantages of DSE presented in this chapter are the main motivation for the work developed in this thesis. The first one is related to the use of abstract summaries. Although abstracting code has the potential to speed up code analysis and produce complete summaries in some cases one might benefit from diving into the abstracted code in order to conclude equivalence. The second drawback presented is related to how the previous approach analyzes the summaries to conclude equivalence. We present here an example where the DSE approach does not classify the versions equivalent although it intuitively should.

3.1 Interpretation Problem

The idea of using abstract summaries to replace common code between two versions has the potential to improve the symbolic execution by reducing the amount of computation performed. Additionally, it allows us to have complete method summaries. However, in some cases, diving into the abstracted code may allow us to produce a more precise differential summary. Consider the code example in Figure 3.1 as version 3 of $\texttt{foo}$ implemented in Figure 2.1. The difference between version 1 in Figure 2.1 and version 3 is that on the latter we return $\texttt{glob}$ instead of $\texttt{var}$. 
public static int foo(int var) {
if (var > 0) {
    if (glob > 0) var = var * var;
    else var = var * (var + 1);
    glob = var;
    return glob;
} else {
    glob = 0;
    return var;
}
}

Figure 3.1: Version 3 of \texttt{foo} introduced in Figure 2.1. In gray is the common code block to be abstracted during analysis.

Following the same strategy presented in Section 2.2, that is using abstract summary to replace the common code (same as before) will generate the following method summary:

$$\texttt{foo3} = \{(V > 0 \land UNTP_1(V,G), \texttt{glob} = \texttt{glob}_1(V,G) \land RETURN = \texttt{glob}_1(V,G)),
\neg(V > 0), \texttt{glob} = 0 \land RETURN = V)\}$$

Looking at the summaries respective to versions 1 (\texttt{foo1}) and version 3 (\texttt{foo3}) we see that they are not partition-effects equivalent. Hence, the analysis described for summaries \texttt{foo1} and \texttt{foo3} would not be able to conclude functional equivalence. However, carefully analyzing both versions we can see that they are in reality functionally equivalent, that is because \texttt{glob} and \texttt{var} are assigned to the same value in both versions at the end of the abstracted code. This failure happens because when abstracting the code block we lose the execution of the instruction \texttt{glob} = \texttt{var} and that would imply the other two partition-effects pair are equivalent. Therefore, ideally, one should be able to use some of the information from the abstracted code if needed in order to entail functional equivalence.
3.2 Elimination Problem

The other drawback of the DSE approach concerns how method summaries are characterized as equivalent. In their approach, they use the notion of partition-effect equivalence. Essentially they look for pairs of equivalent partition-effect pairs, one from each summary and if they all can be paired up the summaries are equivalent. This idea is very useful and its strength is presented in their work. However, when the code changes drastically it might be the case that looking for equivalence is not good enough. In this section, we present a code example where the versions are not partition-effect equivalent but are indeed functionally equivalent.

Consider version 4 of the function \texttt{foo} in Figure 3.2. Comparing it with version 1 in Figure 2.1 one concludes the two versions have the same behavior, that is the return value is the same and the variable \texttt{glob} is assigned to the same value.

```java
public static int foo(int var) {
    if (var < 0) {
        glob = 0;
    } else {
        if (glob > 0) var = var*var;
        else var = var*(var+1);
        glob = var;
    }
    return var;
}
```

\textbf{Figure 3.2:} Version 4 of \texttt{foo} introduced in Figure 2.1. In gray is the common code block to be abstracted during analysis.

Moreover comparing their respective method summaries \texttt{foo1} and \texttt{foo4} generated using the DSE approach without abstracting any code we conclude they are
functionally equivalent:

$$foo1 = \{(V > 0 \land G > 0, GLOB = V \ast V \land RETURN = V \ast V),$$

$$\quad (V > 0 \land (G > 0), GLOB = V \ast (V + 1) \land RETURN = V \ast (V + 1)),$$

$$\quad (! (V > 0), GLOB = 0 \land RETURN = V)\}$$

$$foo4 = \{(V < 0, GLOB = 0 \land RETURN = V),$$

$$\quad (! (V < 0) \land G > 0, GLOB = V \ast V \land RETURN = V \ast V),$$

$$\quad (! (V < 0) \land (G > 0), GLOB = V \ast (V + 1) \land RETURN = V \ast (V + 1))\}$$

where $Write(foo) = \{glob\}$, $GLOB$ refers to the effect of $foo$’s execution on the global variable $glob$, and $V$ is the initial symbolic value of the input $var$.

Investigating the method summaries one can see that there does not exist a pair of equivalent partition-effect pairs, one of each version’s summary. Therefore, these versions are not partition-effect equivalent, and current DSE approach would fail to characterize them as functionally equivalent even though they are.
Chapter 4

Approach

In this chapter, we propose a technique to overcome the drawbacks presented in the previous chapter. This technique is composed of two elements, one for each of the motivating examples mentioned. The first one attempts to dive into the abstracted code if needed to further reduce the size of a method summary. The other proposes a different notion of program equivalence to be used for generating the final table.

4.1 Interpretation

As mentioned in Section 3.1, in some cases, the summaries produced by DSE using uninterpreted functions to abstract common code lose information useful for producing more accurate tables. The idea to overcome such a problem is to first generate a summary for the abstracted piece of code and compose it with the partition-effect pair if needed.

In order to illustrate this idea consider versions 1 and 3 of the method $\texttt{foo}$ in Figure 3.1 and in Figure 2.1 respectively. Following the DSE approach we abstract the code block highlighted in gray and insert the uninterpreted constraint function $\texttt{UNT P}_1$ and the uninterpreted value functions $\texttt{glob}_1$ and $\texttt{var}_1$. However, in our approach, instead of the functions taking as arguments only the read values they also take as arguments the values of the variables written by the abstracted code that are alive before the code is executed. The reason why we also take as arguments the values of written variables is to not lose the information of their previous values.
This way, generating the following summaries $foo_1$ and $foo_2$.

$$
foo_1 = \{(V > 0 \land UNTP_1(V, G), glob = glob_1(V, G) \land RETURN = var_1(V, G)), \\
(\neg (V > 0), glob = 0 \land RETURN = V)\}
$$

$$
foo_3 = \{(V > 0 \land UNTP_1(V, G), glob = glob_1(V, G) \land RETURN = glob_1(V, G)), \\
(\neg (V > 0), glob = 0 \land RETURN = V)\}
$$

Differently from before, we also generate a method summary $unt p_1$ referent to the abstracted code. This summary would be generated using Symbolic Execution by considering the piece of code as an isolated method. Figure 4.1 shows the method created by this approach.

```java
public static void untp1(int var1, int glob1) {
    glob = glob1;
    var = var1;
    if (glob > 0) var = var * var;
    else var = var * (var + 1);
    glob = var;
}
```

**Figure 4.1**: Fictious method $untp1$ whose summary is used in the interpretation algorithm.

Observe that this method takes as input all variables that are read and written by its code block. The written values as input are necessary because we do not want to lose their values prior to the execution of the abstracted code in case there is a path where not all written variables are changed. Following the DSE approach we arrive to the method summary for $untp1$ below

$$
untp1 = \{(G1 > 0, GLOB = V1 * V1 \land VAR = V1 * V1), \\
(\neg (G1 > 0), GLOB = V1 * (V1 + 1) \land VAR = V1 * (V1 + 1))\}
$$
where $G_1$ and $V_1$ are the initial values of $\text{glob}$ and $\text{var}$ passed as arguments to untp1. Additionally, $\text{GLOB}$ and $\text{VAR}$ represent the effects related to the written variables $\text{glob}$ and $\text{var}$.

Let $G(\vec{v})$ be an uninterpreted constraint function where $\vec{v}$ represents all the symbolic inputs, $g$ the associated symbolic summary, and $p$ a partition-effect pair containing an application of $G$. The interpretation algorithm described in Algorithm 1 starts by iterating over the partition-effect pairs of $g$ (line 2). For each partition-effects pair $(i, e)$ in $g$ we first create a copy $p'$ of $p$ and $(i', e')$ of $(i, e)$. Let $G(\vec{a})$ be the application of $G$ in $p'$ then we rewrite $(i', e')$ by replacing each symbolic input $\vec{v}[k]$ by $\vec{a}[k]$ (line 5). After that, we replace the application $G(\vec{a})$ in $p'$ by $i'$ (line 6). In addition, we replace each uninterpreted value function $G_v$ referent to the variable $v \in \text{Write}(g)$ in $p'$ by the value assigned to $v$ in the effects $e'$ lines 7-9. Finally, we check if $p'$ is satisfiable, if it is not we discard it otherwise we add to the new summary (lines 10-12).

---

**Algorithm 1: Interpretation Algorithm**

- **Data:** partition-effects $p$ containing an application of $G$, uninterpreted constraint function $G(\vec{v})$ and its respective summary $g$

- **Output:** A set of partition-effects pairs corresponding to the interpretation of $G$ in $p$

1. $\text{result} \leftarrow \{\}$;
2. for $(i, e) \in g$ do
3.     $p' \leftarrow p\text{.copy}()$;
4.     $(i', e') \leftarrow (i, e)\text{.copy}()$;
5.     $(i', e').\text{rewrite}(\vec{v}, \vec{a})$;
6.     $p'.\text{replace\_all}(G(\vec{a}), i')$;
7.     for $v \in \text{Write}(g)$ do
8.         $p'.\text{replace\_all}(G_v(\vec{a}), e'[v])$; /* $e'[v]$ = effect on $v$ */
9.     end
10.    if $p'$ is satisfiable then
11.        $\text{result}.\text{add}(p')$;
12.    end
13. end
14. return $\text{result}$;

---

The algorithm describes the interpretation of one partition effect containing an
application of $G$. In order to interpret an entire summary $m$ we would run this algorithm for each partition-effects pair $p$ in $m$ containing application of $G$, unite all the resulting sets, and add all elements of $m$ that did not contain application of $G$. Using the interpretation algorithm to interpret the uninterpreted constraint function $UNT P_1$ in the method summary $foo1$ and $foo3$ produces the following method summaries.

$$foo1 = \{(V > 0 \land G > 0, GLOB = V \ast V \land RETURN = V \ast V),$$
$$ (V > 0 \land !(G > 0), GLOB = V \ast (V + 1) \land RETURN = V \ast (V + 1))),$$
$$ (!!(V > 0), glob = 0 \land RETURN = V)\}$$

$$foo3 = \{(V > 0 \land G > 0, GLOB = V \ast V \land RETURN = V \ast V),$$
$$ (V > 0 \land !(G > 0), GLOB = V \ast (V + 1) \land RETURN = V \ast (V + 1))),$$
$$ (!!(V > 0), glob = 0 \land RETURN = V)\}$$

After interpretation we see that versions 1 and 3 of $foo$ are actually partition-effects equivalent, thus functionally equivalent.

4.2 Elimination

As explained in Section 2.2, the previous approach tries to check program equivalence by looking at equivalent pairs of partition-effects pair, one from each summary. As shown in Section 3.2, looking at partition-effects equivalence alone is not enough to conclude two programs are not functionally equivalent, that is, the example showed that in some cases two programs will be functionally equivalent but not partition-effects equivalent.

In this thesis, we propose an approach for analyzing method summaries based on a different notion of equivalence. Generally speaking, the idea is to remove a subset of the input space from one method summary if it is fully accounted for by a unique partition-effects pair from the other summary. Additionally, we remove its partition of the input space from the other method summary as well. This idea
is formalized below as a rule to remove partition-effects pairs.

**Rule 1 (Elimination Rule).** Given two summaries \( s_1 \) and \( s_2 \) and two pairs of partition-effects \( pe_1 = (i_1, e_1) \) and \( pe_2 = (i_2, e_2) \), where \( pe_1 \in s_1 \) and \( pe_2 \in s_2 \) such that

\[
i_1 \land e_1 \implies i_2 \land e_2
\]

then we remove \( pe_1 \) from \( s_1 \). If \( (i_2 \land \lnot i_1) \land e_2 \) is unsat we also remove \( pe_2 \) from \( s_2 \) otherwise we update \( pe_2 \) to \( (i_2 \land \lnot i_1, e_2) \). For summaries \( s_1 \) and \( s_2 \), if there exists such partition-effects pairs \( pe_1 \in s_1 \) and \( pe_2 \in s_2 \) we denote the resulting pair of summaries after elimination rule application by \( \tau(pe_1, s_1, pe_2, s_2) \), which is a pair of summaries the first one being \( s_1 \setminus \{pe_1\} \) and the other the updated \( s_2 \).

In this project, we use the elimination rule to classify methods as functionally equivalent. In this thesis, we analyze summaries seeking algorithmic equivalence which essentially means that exhaustively applying the elimination rule produces empty summaries.

**Definition 6 (Algorithmic Equivalence).** Two methods \( m, m' \) with respective symbolic summaries \( s \) and \( s' \) are algorithmically equivalent if and only if there exists a finite sequence \( \{(r_i, s_i)\}_{0 \leq i \leq n} \) for some natural number \( n \) such that \( (s, s') = (r_0, s_0), (r_n, s_n) = (\emptyset, \emptyset) \) and for all \( 0 \leq i < n \) there exists \( p_i \in r_i \) and \( q_i \in s_i \) such that we have either \( \tau(p_i, r_i, q_i, s_i) = (r_{i+1}, s_{i+1}) \) or \( \tau(q_i, s_i, p_i, r_i) = (r_{i+1}, s_{i+1}) \).

Algorithmic Equivalence is finer-grained than functional equivalence and coarser-grained than partition-effect equivalence. Two partition-effects equivalent methods are algorithmically equivalent, however, the inverse is not necessarily true (Theorem A.1.2 in the Appendix A). Similarly, two algorithmically equivalent methods are functionally equivalent however the inverse is not true (Theorem A.2.3 in the Appendix A).

Following those definitions we can prove that versions 1 and 4 of the method \( \text{foo} \) are functionally equivalent. Considering their respective summaries \( \text{foo}1 \) and \( \text{foo}4 \) we can apply the elimination rule until we get two empty summaries, concluding algorithmic equivalence and thus functional equivalence.
4.3 General Algorithm

The idea of the general algorithm is to combine the elimination rule and the interpretation algorithm until we reach a fixed point or both summaries are empty.

Algorithm 2: General Algorithm

Data: summaries $m$ and $m'$

1. $\text{done} \leftarrow \text{false}$;
2. while (not done) and ($m$.size() + $m'$.size() > 0) do
   3.   $\text{done} \leftarrow \text{true}$;
   4.   for $(i, e), (i', e') \in m \times m'$ do
      5.     if $i \land e \implies i' \land e'$ then
         6.       $m$.remove($(i, e))$;
         7.       $m'$.remove($(i', e'))$;
         8.       if $\neg i \land i' \land e'$ is sat then
            9.         $m'$.add($(i' \land \neg i, e'))$;
         end
      10.    done $\leftarrow \text{false}$;
      11. else if $i' \land e' \implies i \land e$ then
           12.       $m$.remove($(i, e))$;
           13.       $m'$.remove($(i', e'))$;
           14.       if $\neg i' \land i \land e$ is sat then
              15.         $m$.add($(i \land \neg i', e))$;
           end
           16.    done $\leftarrow \text{false}$;
      end
   17.  end
   18. if done then
      19.    if there is still functions to be interpreted then
         20.      interpret_one($m, m'$); /* interprets exactly one */
      21.    end
   22. end
   23. done $\leftarrow \text{false}$;
24. end
25. end

In Algorithm 2 we show the complete algorithm for differentiating two summaries $m$ and $m'$. We start by iterating over all elements of the cross product $m \times m'$ until we find an element $((i, e), (i', e'))$ such that $i \land e \implies i' \land e'$ or $i' \land e' \implies i \land e$. 
(line 5 and 12). When such a pair is found we apply the elimination rule for those pairs accordingly then we start over with the updated summaries. Eventually, either one of the summaries becomes empty, in which case the algorithm terminates, or there is no such pair anymore (done is true in line 21). In the case there is no such pair, we interpret one of the uninterpreted constraint functions (line 23) using the interpretation algorithm described in Algorithm 1 and start over, or if there is no uninterpreted constraint function left to be interpreted we terminate the algorithm.

```java
public static int f(int x) {
    int val = 0;
    if (x <= 0) val = 0;
    else if (x < 10) val = x * x;
    else if (x < 30) val = (20 - x) * (20 - x);
    else val = (40 - x) * (40 - x);
    if (x > 40) state = 1;
    else if (x%10 == 0 || x <= 0) state = 0;
    else if ((x/10)%2 == 0) state = 1;
    else state = -1;
    return state * val;
}
```

Consider the two implementations of the function `f` in Figure 4.2. Let us look at how the algorithm works in practice. First, we abstract the gray block of code using our approach inserting the uninterpreted constraint function $U_1$ and the uninterpreted value function $state_1$. Secondly, we generate summaries for each version.

**Figure 4.2:** Implementations of $f$. Version 1 on the left hand side and Version 2 on the right hand side. In gray is a common code block to be abstracted during analysis.
of \( f \) and for the abstracted code, respectively, \( f_1, f_2 \) and \( u \) as follows:

\[
\begin{align*}
u &= \{ \\
(X_1 > 40, & STATE = 1), \\
(! (X_1 > 40) \land (X \mod 10 = 0) \| X_1 < 0), & STATE = 0), \\
(! (X_1 > 40) \land !((X \mod 10 = 0) \| X_1 < 0)) \land ((X_1 / 10) \mod 2 = 0), & STATE = 1), \\
(! (X_1 > 40) \land ! (X_1 < 10) \land !((X_1 / 10) \mod 2 = 0), & STATE = -1).
\}
\end{align*}
\]

\[
f_1 = \{ \\
(X \leq 0, & U_1(X, S), \\
STATE = state_1(X, S) \land RETURN = 0), \\
(! (X \leq 0) \land (X < 10) \land U_1(X, S), \\
STATE = state_1(X, S) \land RETURN = X^2 \ast state_1(X, S)), \\
(! (X \leq 0) \land ! (X < 10) \land (X < 30) \land U_1(X, S), \\
STATE = state_1(X, S) \land RETURN = (20 - X)^2 \ast state_1(X, S)), \\
(! (X \leq 0) \land ! (X < 10) \land !(X < 30) \land U_1(X, S), \\
STATE = state_1(X, S) \land RETURN = (40 - X)^2 \ast state_1(X, S)).
\}
\]

\[(4.1)\]
\[ f_2 = \{ \]
\[ (X \leq 0 \land U_1(X, S), \]
\[ STATE = 0 \land RETURN = 0), \]
\[ (!X \leq 0) \land (X < 40) \land ((X/10) \mod 2 = 0) \land U_1(X, S), \]
\[ STATE = state_1(X, S) \land RETURN = (X - 10 \ast (X/10))^2 \ast state_1(X, S)), \]
\[ (!X \leq 0) \land (X < 40) \land ((X/10) \mod 2 = 0) \land U_1(X, S), \]
\[ STATE = state_1(X, S) \land RETURN = (X - 10 \ast (1 + X/10))^2 \ast state_1(X, S)), \]
\[ (!X \leq 0) \land !(X < 40) \land U_1(X, S), \]
\[ STATE = state_1(X, S) \land RETURN = (X - 40)^2 \ast state_1(X, S)) \]
\} 

(4.2)

In order to illustrate better the algorithm lets rename the elements of the above summaries as

\[ f_1 = \{(i_{11}, e_{11}), (i_{12}, e_{12}), (i_{13}, e_{13}), (i_{14}, e_{14})\} \]
\[ f_2 = \{(i_{21}, e_{21}), (i_{22}, e_{22}), (i_{23}, e_{23}), (i_{24}, e_{24})\} \]

where the order in which the elements appear in the sets is the same as the one in Equation 4.1 and Equation 4.2 and each \((i_{rs}, e_{rs}) = p_{rs}\).

The algorithm described above would follow the following steps:

**Step 1.** \(i_{12} \land e_{12} \implies i_{22} \land e_{22}\) and \(\neg i_{12} \land i_{22} \land e_{22}\) is sat

\[ f_1 \rightarrow s_0 = \{(i_{11}, e_{11}), (i_{13}, e_{13}), (i_{14}, e_{14})\} \]
\[ f_2 \rightarrow s_0' = \{(i_{21}, e_{21}), (i_{22} \land \neg i_{12}, e_{22}), (i_{23}, e_{23}), (i_{24}, e_{24})\} \]

**Step 2.** \((i_{22} \land \neg i_{12}) \land e_{22} \implies i_{13} \land e_{13}\) and \((i_{22} \land \neg i_{12}) \land i_{13} \land e_{13}\) is sat

22
\[ s_0 \rightarrow s_1 = \{ (i_{11}, e_{11}), (i_{13} \land \neg (i_{22} \land \neg i_{12}), e_{13}), (i_{14}, e_{14}) \} \]

\[ s'_0 \rightarrow s'_1 = \{ (i_{21}, e_{21}), (i_{23}, e_{23}), (i_{24}, e_{24}) \} \]

**Step 3.** \((i_{13} \land \neg (i_{22} \land \neg i_{12})) \land e_{13} \implies i_{23} \land e_{23}\) and \(\neg (i_{13} \land \neg (i_{22} \land \neg i_{12})) \land i_{23} \land e_{23}\) is sat

\[ s_1 \rightarrow s_2 = \{ (i_{11}, e_{11}), (i_{14}, e_{14}) \} \]

\[ s'_1 \rightarrow s'_2 = \{ (i_{21}, e_{21}), (i_{23} \land \neg (i_{13} \land \neg (i_{22} \land \neg i_{12})), e_{23}), (i_{24}, e_{24}) \} \]
Step 4. \((i_{23} \land \neg(i_{13} \land \neg(i_{22} \land \neg i_{12}))) \land e_{23} \implies i_{14} \land e_{14}\) and \(\neg(i_{23} \land \neg(i_{13} \land \neg(i_{22} \land \neg i_{12}))) \land i_{14} \land e_{14}\) is sat

\[s_2 \rightarrow s_3 = \{(i_{11}, e_{11}), (i_{14} \land \neg(i_{23} \land \neg(i_{13} \land \neg(i_{22} \land \neg i_{12}))), e_{14}\}\]

\[s'_2 \rightarrow s'_3 = \{(i_{21}, e_{21}, (i_{24}, e_{24})\}\]

Step 5. \((i_{14} \land \neg(i_{23} \land \neg(i_{13} \land \neg(i_{22} \land \neg i_{12}))) \land e_{14} \implies i_{24} \land e_{24}\) and \(\neg(i_{14} \land \neg(i_{23} \land \neg(i_{13} \land \neg(i_{22} \land \neg i_{12}))) \land i_{24} \land e_{24}\) is unsat

\[s_3 \rightarrow s_4 = \{(i_{11}, e_{11})\}\]

\[s'_3 \rightarrow s'_4 = \{(i_{21}, e_{21})\}\]

After Step 5, there are no more pairs to apply the elimination rule, then the algorithm will check if there is uninterpreted constraint functions left to interpret. In our case there is \(U_1\). In our example, the clause \(X \leq 0\) in \(i_{11}\) and in \(i_{21}\) will make most of the copies to be discarded since they will be unsat.

Step 6. Interpreting \(U_1\)

\[s_4 \rightarrow s_5 = \{((X \leq 0 \land \neg(X > 40)) \land (X \mod 10 = 0) \land X \leq 0), STATE = 0 \land RETURN = 0\}\]

\[s'_4 \rightarrow s'_5 = \{((X \leq 0 \land \neg(X > 40)) \land (X \mod 10 = 0) \land X \leq 0), STATE = 0 \land RETURN = 0\}\]

Then the algorithm goes back to attempt to use the elimination rule, renaming \(s_5\) as \((i_5, e_5)\) and \(s'_5\) as \((i'_5, e'_5)\) and the next step goes as follows:

Step 7. \(i_5 \land e_5 \implies i'_5 \land e'_5\) and \(\neg i_5 \land i'_5 \land e'_5\) is unsat

\[s_5 \rightarrow s_6 = \{}\]

\[s'_5 \rightarrow s'_6 = \{}\]
Hence terminating the algorithm and concluding the two versions of the method \( f \) are equivalent.
Chapter 5

Implementation

In this chapter, we discuss the implementation of a prototype tool Diffel for checking the approach presented in this thesis. Our approach for analyzing two versions of the same method requires one to first get hold of a method summary in the format of a set of partition-effects pairs. We build our tool on top of the Java PathFinder symbolic execution framework (symbc JPF) [12]. In the following sections, we explain how we adapted SPF to generate method summaries as well as the adaptations in the source code in order to be suited for analysis.

5.1 symbc JPF

The JPF symbolic execution framework is an extension of JPF, referred to as the symbc JPF extension. It includes a custom bytecode instruction factory that modifies or extends the standard concrete execution semantics with symbolic execution. Symbolic information is stored as attributes associated with program data. JPF allows one to choose from several constraint programming systems for decision procedure support and constraint solving. Due to the inherent undecidability of state matching when states represent path conditions on unbounded data, JPF refrains from performing state matching on symbolic values. To address loops and recursion, JPF can be constrained by setting a depth limit on its search or by restricting the number of constraints encoded for a specific path. JPF signals when it reaches one of these constraints during symbolic execution.
We have modified the symbe JPF to incorporate our approach. Although symbe JPF lets one specify which constraint solving to be used in this prototype tool the user is required to choose z3 for handling all decision procedures and constraint solving. z3 supports queries for validity and satisfiability over logical formulae expressed in terms of an extensive set of basic types, including uninterpreted functions and user-defined types. Furthermore, we used the z3 java library to implement the algorithm presented in this thesis.

In order to check equivalence and reduce the method summaries, we have implemented a Driver java class that reads method summaries produced by the customized JPF symbolic execution listener. The method summaries generated consist essentially of a list of partition-effects objects and some helper methods. Similarly, we have encoded partition-effects as a Java object consisting of a z3 formula for the input constraint, a Java Map for the effects beside several other helper methods and attributes to aid in performing the algorithms.

5.1.1 Uninterpreted functions

A significant part of this project consists of the use of uninterpreted functions to abstract away common code. The extension symbe JPF does not provide any support for defining uninterpreted functions. Therefore, in order to be able to generate summaries with uninterpreted functions in them we extended symbe JPF enabling uninterpreted function applications as expressions. For this prototype uninterpreted functions’ signatures consist only of numeric types and booleans, that is int, long, short, double, float, boolean.

In symbe JPF each function call is executed by the symbolic execution engine as if the code was inline. When executing a function call instruction \( a = f(b) \) it keeps updating the path condition and exploring all the paths within method’s \( f \) definition until reaching the return statement and assigning the an expression referring to the returned value to \( a \). We modified the listener to assign uninterpreted function applications instead of the actual return value.
Figure 5.1: Illustration of modifications made in order to allow uninterpreted function applications as expressions in symbolic JPF. We have the application of two uninterpreted functions `UNTP_C_1` and `UNTP_I_1_STATE`.

Consider the piece of code in Figure 5.1. Normally, in symbolic JPF, when line 11 is reached the `UNTP_C_1` would be executed evaluating the call to the return value `true` and therefore never even getting to line 12. However, we modified the symbolic listener so `UNTP_C_1` is treated as an uninterpreted function, thus, we update the path condition `P` to `P ∧ ¬UNTP_C_1(v[x], v[GLOB_STATE])`, where `v[a]` represent the symbolic expression assigned to the variable `a` at that exploration instant. Analogously, the value of the variable `GLOB_STATE` would get the value `UNTP_I_1_STATE(v[x], v[GLOB_STATE])` instead of the returned value `0`.

In our prototype, uninterpreted functions are identified by the symbolic listener using string matching on the method name. In future work, we intend to make this process more modular and better suited for end users.

5.1.2 Effects

Another crucial part of our approach is accounting for the effects as well. We modified the symbolic listener to keep track of the symbolic expressions assigned
to variables in the Write set of a method. Each time a variable assignment is executed we check if the variable is in Write if it is we update a Java Map object keeping track of it. In this prototype, the symbolic listener checks if a variable is in the Write set of a method by string matching on the variable name. Before analysis, we edit the code adding the prefix GLOB to a variable if it needs tracking. Figure 5.2 shows a method implementation before any modifications and in Figure 5.3 the modified version of it in order to be analyzed.

5.2 Tool input

In this section we present some of the important specifications the two methods must have in order to be analyzed by our prototype tool. As mentioned before, the prototype tool strongly relies on the naming of the method’s code elements. We explain here how the uninterpreted functions and effects’ variables must be formatted.

```java
public static int f(int x) {
    int val = 0;
    if (x <= 0) val = 0;
    else if (x < 10) val = x * x;
    else if (x < 30) val = (20 - x) * (20 - x);
    else val = (40 - x) * (40 - x);

    if (x > 40) state = 1;
    else if (x % 10 == 0 || x <= 0) state = 0;
    else if ((x / 10) % 2 == 0) state = 1;
    else state = -1;

    return state * val;
}
```

Figure 5.2: Code snippet before any modifications are made to serve as input for our tool. In gray is the code block to be abstracted.
public static int UNTP_L1_state(int x1, int state1) {
    return 0;
}

public static boolean UNTP_C1(int x1, int state1) {
    return true;
}

public static int f(int x, int state) {
    int GLOB RETURN;
    int GLOB_state = state;
    int val = 0;
    if (x < 0) val = 0;
    else if (x < 10) val = x * x;
    else if (x < 30) val = (20 - x) * (20 - x);
    else val = (40 - x) * (40 - x);
    if (!UNTP_C1(x, GLOB_state)) {
        GLOB_state = UNTP_L1_state(x, GLOB_state);
    }
    GLOB_RETURN = GLOB_state * val;
    return GLOB_state * val;
}

Figure 5.3: Modified version of code snippet in Figure 5.2 to serve as input for our tool. In gray is the replacement for the abstracted code.

5.2.1 Code Abstraction

One of the aspects of our approach is the abstraction of common code and replacing it with uninterpreted functions. The uninterpreted functions are caught by the symbolic listener using string matching on its name. In our approach, each abstraction generates one uninterpreted constraint function and several uninterpreted value functions, one for each written variable. It follows below how one should format them:

**Uninterpreted Constraint Functions.** Each uninterpreted constraint function name starts with the prefix `UNTP_C` and carries a unique integer `id` with it related to the abstracted code, that is, for an abstracted piece of code the generated uninterpreted constraint function and all the uninterpreted value functions carry the same number.
Thus, all uninterpreted constraint functions are in the format \texttt{UNTP.C.id}.

\textbf{Uninterpreted Value Functions.} Each uninterpreted value function name carries three parameters, its type (real or integer), its unique \textit{id}, and the effect variable name \textit{varname} related to it. The type is encoded by the character \texttt{I} for integer or \texttt{R} for real. Furthermore, effect variable names cannot contain \_ \_. Additionally, uninterpreted value functions start with \texttt{UNTP.}. Thus, all uninterpreted value functions are in the format \texttt{UNTP.{|I|R}.id.varname}.

The abstracted code must be replaced by an if-statement where the condition is the negation of the uninterpreted constraint function application and the body is composed by variable-assignment-statements one for each written variable in Figure 5.3 we can see an example of it highlighted in gray.

\textbf{5.2.2 Effects}

Similarly to uninterpreted functions, the variables in the effects are caught by the symbolic listener based on string matching. For written variables, we simply add the prefix \texttt{GLOB.}. In our prototype, we extend the function arguments list with the effect variables as well. The reason for that is that by doing so symbolic JPF will automatically initialize a symbolic value to it. Additionally, we declare all written variables at the top of the method and initialize them using the values passed in the arguments. For simplicity, we also declare a variable called \texttt{GLOB RETURN} that will keep track of the return value.
Chapter 6

Evaluation

The goal of our technique is to characterize the difference between two versions of a method in terms of the induced effects. We aim to specify the exact input space partition in which the two method versions are assured to present different behavior. We evaluate the strength of our approach in comparison to partition-effects equivalence approach by considering the following research questions:

\textit{RQ1.} How effective is our approach in classifying two functionally equivalent programs compared with partition-effects equivalence technique?

\textit{RQ2.} How robust is our approach in restricting the input space capturing only the inputs that expose the difference between versions?

6.1 Dataset

To evaluate our approach we used part of the IntroClass benchmark for Java [6]. The IntroClass benchmark is composed of 6 different kinds of programs and for each kind there are several implementations submitted by students and an oracle. The two kinds used for our evaluation are Median and Smallest. The Median kind consists of implementations of a method that computes the median value among three inputted integers. The Smallest kind consists of implementations of a method that computes the smallest among 4 inputted integers.
In order to perform our analysis we first filtered the implementations. We used the following filters in that order eliminating those respective number of implementations.

- did not take integers as input (-6 Median, -6 Smallest)
- returned answers in a very different format (-3 Median, -3 Smallest)
- were syntactically equivalent to another implementation (-5 Median, -9 Smallest)
- had compilation problems after modifications (-1 Smallest)
- use datatypes not supported by our prototype (-1 Smallest)
- were syntactically equivalent up to variable renaming and submitted by the same student (-2 Smallest)

After filtering the programs we run a script to format them to serve as input to our prototype tool. In the original source code, the methods would prompt a query for the user to input some values and instead of returning the result it would print out a sentence containing the final answer. For this analysis, we modified the programs to take the input as arguments and to return the computed value (the median or the smallest). Additionally, all methods had a class to encapsulate ints we removed those classes and used the built-in type int as is. In the appendix, in Figure B.2 we have an example of one code snippet from the IntroClass benchmark for Java and in Figure B.1 the automatically generated code snippet from it.

One of the aspects of our approach is the abstraction of common code. Ideally, one could use a syntactic code differentiation tool to automatically perform the abstraction, to generate a method respective to the abstracted piece of code for further interpretation and to generate all necessary uninterpreted functions. However, for this prototype, all code abstraction was manually performed obeying the input formatting rules specified in Section 5.2.

In addition, we renamed all implementation classes each one was renamed to the format kind_id, e.g. Median_17_a, Smallest_15, where kind is equal to Median or Smallest and id is, in most cases, a unique natural number representing a student who submitted an implementation. In this benchmark repository,
some students submitted multiple implementations and each submission receives a different id composed of the number associated with the student and a unique letter.

### 6.2 Results

In order to evaluate our approach we conduct three experiments. In the first experiment, abstraction experiment, we compare versions submitted by the same students using code abstraction and uninterpreted functions. In the other two experiments, we do not abstract any code. In the second experiment, oracle experiment, we compare each version against the oracle and in the third, versioning experiment, we compare versions submitted by the same student.

Each run in an experiment consists of generating method summaries for each version using our customized symbc JPF. Then, we process the summaries in four different manners. The structure of the General Algorithm (Section 4.3) is used in all of them, they differ in whether partition-effects are removed using the Elimination Rule (Rule 1) or simple partition-effects equivalence and in whether uninterpreted functions are interpreted or not.

- Ei. This method uses the Elimination Rule and the Interpretation Algorithm.
- E. This method uses Elimination Rule but does not interpret any uninterpreted function.
- I. This method uses the Interpretation Algorithm but remove partition-effects pairs based on equivalence.
- N. This method only eliminates partition-effects pairs from both summaries if they are equivalent and does not interpret any uninterpreted function.

**Metrics.** For each method of processing, once terminated we analyzed the resulting method summaries using two metrics. The first metric (# pes ) is a pair of integers equal to the number of partition-effects in each one of the two final method summaries. In all of the processing strategies empty summaries at termination imply the versions are equivalent. Therefore, this metric indicates the ability
of the processing method to classify two methods as equivalent. By looking at the resulting values on this metric for the experiments we can investigate RQ1.

The second metric (diff %) is computed in three steps. First of, for each of the partition-effects pairs \( p \) in one of the method summaries we get 10 different models as solutions to the input constraint part of \( p \). Secondly, for each one of the models we apply each version to it and check whether the input induces a behavior difference or not. The behavior difference is checked by looking at the effects on each application. Finally, we count the percentage of the models that produced different behavior, arriving at a number between 0 and 1. The idea is that high values on this metric mean that the partition-effects remaining are likely to induce a divergence. Therefore, this metric indicates the ability of the processing model to capture the partition of the input space that will produce different behavior. By looking at the resulting values on this metric for the experiments we can investigate RQ2.

**Interpreting Results.** For metric # pes v1,v2, what matters the most is whether it is equal to \((0,0)\) indicating the processing method classified both versions as equivalent.

For metric diff %, if the method versions are not functionally equivalent, the higher its value the better the processing method captures the partition of the input space that produces different behavior. If the method versions are functionally equivalent diff % is indeed expected to be equal 0.

### 6.2.1 Abstraction Experiment

For this experiment, we first analyzed all versions submitted by the same student and evaluated whether there were enough common lines of code to be abstracted. After checking carefully the implementations we saw a pattern in the versions submitted by the same student. In most of the cases, the changes made were not suited to abstraction, that is changing operators (e.g. change < to <=) or a completely new approach to the problem as if they started from scratch. In the cases with significant common code, we abstracted and added uninterpreted functions.

Table 6.1 shows the results of this experiment. In only one case we do not have
Table 6.1: Results for the abstraction experiment. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0.0. EI processing method achieving the best results supporting the strength of our approach.

Table 6.2 and Table 6.3 show the results of this experiment for Median and Smallest kind respectively. Given that partition-effects equivalence implies algorithmic equivalence it is no surprise to see that E processing method is better than the N processing method in all cases. However, in this experiment, we can see the increase in metric diff %. In some cases the difference is very significant, for example, for MedianExperiment_2.a_b we go from 0.15 to 1. That supports the claim that our approach can make an expressive difference in getting the partition
<table>
<thead>
<tr>
<th>File</th>
<th>E</th>
<th># pes</th>
<th>N</th>
<th>diff %</th>
<th># pes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MedianExperiment_17_c</td>
<td>0.00</td>
<td>5, 5</td>
<td>0.00</td>
<td>6, 5</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_8</td>
<td>0.14</td>
<td>3, 4</td>
<td>0.09</td>
<td>6, 5</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_10</td>
<td>1.00</td>
<td>4, 4</td>
<td>0.42</td>
<td>6, 8</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_6_a</td>
<td>0.90</td>
<td>1, 1</td>
<td>0.08</td>
<td>5, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_35_b</td>
<td>1.00</td>
<td>2, 1</td>
<td>0.18</td>
<td>6, 5</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_2_a</td>
<td>0.80</td>
<td>3, 2</td>
<td>0.23</td>
<td>6, 7</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_31</td>
<td>1.00</td>
<td>2, 2</td>
<td>0.23</td>
<td>4, 9</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_7_a</td>
<td>0.50</td>
<td>2, 2</td>
<td>0.14</td>
<td>5, 9</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_20</td>
<td>0.33</td>
<td>6, 5</td>
<td>0.33</td>
<td>6, 5</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_6_c</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.67</td>
<td>5, 11</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_33</td>
<td>0.51</td>
<td>4, 5</td>
<td>0.30</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_22_a</td>
<td>1.00</td>
<td>2, 2</td>
<td>0.25</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_16</td>
<td>0.90</td>
<td>5, 6</td>
<td>0.61</td>
<td>5, 11</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_2_b</td>
<td>1.00</td>
<td>2, 2</td>
<td>0.15</td>
<td>5, 8</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_30</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_13</td>
<td>0.90</td>
<td>5, 6</td>
<td>0.61</td>
<td>5, 11</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_12</td>
<td>0.00</td>
<td>2, 2</td>
<td>0.00</td>
<td>5, 4</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_34_b</td>
<td>0.95</td>
<td>1, 1</td>
<td>0.06</td>
<td>5, 9</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_28</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_2_c</td>
<td>0.80</td>
<td>3, 2</td>
<td>0.23</td>
<td>6, 7</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_34_a</td>
<td>0.95</td>
<td>1, 1</td>
<td>0.06</td>
<td>5, 9</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_9</td>
<td>0.90</td>
<td>5, 6</td>
<td>0.61</td>
<td>5, 11</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_21_a</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_3</td>
<td>0.66</td>
<td>2, 1</td>
<td>0.18</td>
<td>5, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_18</td>
<td>0.63</td>
<td>2, 1</td>
<td>0.18</td>
<td>5, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_23_b</td>
<td>0.90</td>
<td>3, 4</td>
<td>0.57</td>
<td>3, 8</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_11</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_4_b</td>
<td>0.40</td>
<td>3, 2</td>
<td>0.20</td>
<td>5, 5</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_25</td>
<td>0.66</td>
<td>2, 1</td>
<td>0.18</td>
<td>5, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_14</td>
<td>0.66</td>
<td>2, 1</td>
<td>0.18</td>
<td>5, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_19</td>
<td>0.50</td>
<td>2, 2</td>
<td>0.00</td>
<td>6, 8</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_32</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_22_b</td>
<td>0.09</td>
<td>5, 6</td>
<td>0.08</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_17_b</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_35_a</td>
<td>0.20</td>
<td>3, 2</td>
<td>0.08</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_17_a</td>
<td>0.09</td>
<td>5, 6</td>
<td>0.08</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_26</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_4_a</td>
<td>0.00</td>
<td>2, 2</td>
<td>0.00</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_24</td>
<td>0.90</td>
<td>4, 4</td>
<td>0.60</td>
<td>5, 7</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_23_a</td>
<td>0.90</td>
<td>3, 3</td>
<td>0.41</td>
<td>4, 9</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_1_b</td>
<td>0.90</td>
<td>5, 7</td>
<td>0.63</td>
<td>5, 12</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_15_a</td>
<td>1.00</td>
<td>4, 3</td>
<td>0.50</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>MedianExperiment_15_b</td>
<td>0.90</td>
<td>1, 1</td>
<td>0.08</td>
<td>5, 6</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.2:** Oracle Experiment results restricted to the Median kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0.0. This table shows a significant improvement on metric diff % for E supporting the strength in restricting the input space.
<table>
<thead>
<tr>
<th>File</th>
<th>E</th>
<th>N</th>
<th>diff %</th>
<th># pes</th>
<th>diff %</th>
<th># pes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallestExperiment_7_c</td>
<td>0.00</td>
<td>4, 2</td>
<td>0.00</td>
<td>8, 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_21_a</td>
<td>0.34</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_15</td>
<td>0.32</td>
<td>6, 7</td>
<td>0.15</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_20</td>
<td>0.33</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_22</td>
<td>0.53</td>
<td>6, 7</td>
<td>0.45</td>
<td>8, 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_8</td>
<td>0.32</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_17</td>
<td>0.34</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_25</td>
<td>0.90</td>
<td>6, 10</td>
<td>0.37</td>
<td>7, 28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_13_a</td>
<td>0.58</td>
<td>5, 12</td>
<td>0.43</td>
<td>8, 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_7_b</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_12</td>
<td>0.33</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_5</td>
<td>0.34</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_31</td>
<td>0.00</td>
<td>6, 5</td>
<td>0.00</td>
<td>8, 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_4_a</td>
<td>0.18</td>
<td>8, 9</td>
<td>0.15</td>
<td>8, 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_11_b</td>
<td>0.71</td>
<td>6, 12</td>
<td>0.33</td>
<td>8, 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_29</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_26</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_4_b</td>
<td>0.33</td>
<td>6, 7</td>
<td>0.14</td>
<td>8, 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_11_a</td>
<td>0.96</td>
<td>8, 15</td>
<td>0.81</td>
<td>8, 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_23</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_16_a</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_18_c</td>
<td>0.80</td>
<td>6, 4</td>
<td>0.50</td>
<td>8, 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_7_a</td>
<td>0.00</td>
<td>4, 2</td>
<td>0.00</td>
<td>8, 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_16_b</td>
<td>0.63</td>
<td>8, 8</td>
<td>0.53</td>
<td>8, 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_13_b</td>
<td>0.84</td>
<td>6, 20</td>
<td>0.68</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_2</td>
<td>0.76</td>
<td>6, 11</td>
<td>0.40</td>
<td>8, 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_18_b</td>
<td>0.23</td>
<td>7, 6</td>
<td>0.12</td>
<td>8, 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_16_e</td>
<td>0.18</td>
<td>8, 9</td>
<td>0.15</td>
<td>8, 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_18_e</td>
<td>0.23</td>
<td>8, 7</td>
<td>0.21</td>
<td>8, 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallestExperiment_3_a</td>
<td>0.03</td>
<td>6, 4</td>
<td>0.05</td>
<td>8, 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Oracle Experiment results restricted to the Smallest kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0.0. This table shows a significant improvement on metric diff % for E supporting the strength in restricting the input space.
of the input space that exposes the behavioral difference among versions. On the other hand, we could not see much difference concerning the ability of proving functionally equivalence. For this experiment, there were no cases where either approach proved functional equivalence.

### 6.2.3 Versioning Experiment

In this experiment, we compare versions submitted by the same student that do not have a common code block worth being abstracted. Similarly to Oracle Experiment, in this experiment there was no code abstraction, therefore the resulting tables show only E and N processing methods. Through this experiment we can study the strength of the elimination rule and algorithmic equivalence with respect to partition-effects equivalence for methods that are produced by the same developer and where the versions do not have a large common code block.

<table>
<thead>
<tr>
<th>File</th>
<th>E diff %</th>
<th># pes</th>
<th>N diff %</th>
<th># pes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MedianExperiment_34_a_b</td>
<td>—</td>
<td>0,0</td>
<td>—</td>
<td>0,0</td>
</tr>
<tr>
<td>MedianExperiment_35_a_b</td>
<td>0.00</td>
<td>2,2</td>
<td>0.00</td>
<td>6,5</td>
</tr>
<tr>
<td>MedianExperiment_17_a_b</td>
<td>0.91</td>
<td>6,8</td>
<td>0.67</td>
<td>6,13</td>
</tr>
<tr>
<td>MedianExperiment_2_a_c</td>
<td>—</td>
<td>0,0</td>
<td>—</td>
<td>0,0</td>
</tr>
<tr>
<td>MedianExperiment_6_a_c</td>
<td>0.90</td>
<td>5,7</td>
<td>0.63</td>
<td>6,11</td>
</tr>
<tr>
<td>MedianExperiment_17_c_b</td>
<td>0.90</td>
<td>5,8</td>
<td>0.65</td>
<td>5,13</td>
</tr>
<tr>
<td>MedianExperiment_2_c_b</td>
<td>1.00</td>
<td>2,1</td>
<td>0.16</td>
<td>5,7</td>
</tr>
<tr>
<td>MedianExperiment_15_a_b</td>
<td>1.00</td>
<td>3,4</td>
<td>1.00</td>
<td>3,4</td>
</tr>
</tbody>
</table>

Table 6.4: Versioning experiment results restricted to the Median kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % supporting it strength in restricting the input space.

Table 6.4 and Table 6.5 show the results of this experiment for Median and Smallest kind respectively. Similarly to the Oracle Experiment the E processing method is better than the N processing method in all cases. For the Median kind we can see the results achieved by E processing method were very good, diff % values are at least 0.9 which supports the ability of our approach to characterize the input space generating behavioral difference for software written by the same developer.
<table>
<thead>
<tr>
<th>File</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>N</td>
</tr>
<tr>
<td>SmallestExperiment_18_b_e</td>
<td>0.60</td>
<td>8, 7</td>
</tr>
<tr>
<td>SmallestExperiment_16_c_b</td>
<td>0.41</td>
<td>10, 10</td>
</tr>
<tr>
<td>SmallestExperiment_7_c_b</td>
<td>0.90</td>
<td>5, 10</td>
</tr>
<tr>
<td>SmallestExperiment_18_c_b</td>
<td>0.30</td>
<td>7, 7</td>
</tr>
<tr>
<td>SmallestExperiment_4_a_b</td>
<td>0.30</td>
<td>9, 11</td>
</tr>
<tr>
<td>SmallestExperiment_16_a_b</td>
<td>0.77</td>
<td>17, 10</td>
</tr>
<tr>
<td>SmallestExperiment_7_a_c</td>
<td>—</td>
<td>0, 0</td>
</tr>
<tr>
<td>SmallestExperiment_13_a_b</td>
<td>0.90</td>
<td>5, 10</td>
</tr>
<tr>
<td>SmallestExperiment_11_a_b</td>
<td>0.83</td>
<td>14, 23</td>
</tr>
<tr>
<td>SmallestExperiment_16_a_c</td>
<td>0.71</td>
<td>16, 9</td>
</tr>
<tr>
<td>SmallestExperiment_7_a_b</td>
<td>0.90</td>
<td>5, 10</td>
</tr>
</tbody>
</table>

Table 6.5: Versioning experiment results restricted to the Smallest kind. The higher the value of diff % the better, while for # pes it matters only when it is equal to 0,0. This table shows a significant improvement on metric diff % supporting it strength in restricting the input space.

developer. However, the experiment on Smallest did not show much difference from the previous experiment. Therefore, further analysis would be required to infer more reliable conclusions.

6.3 RQ1: Proving Equivalence

RQ1. How effective is our approach in classifying two functionally equivalent programs compared with simple partition-effects equivalence technique?

As we have mentioned before, partition-effects equivalence implies algorithmic equivalence, therefore, our approach can prove equivalence in all cases that using the notion of partition-effects equivalence can. However, in our experiments, there was only one example (MedianExperimentUnint_17_a_c) where our approach proved equivalence and an approach using partition-effects equivalence did not. Therefore, through those experiments we cannot conclude that in practice our approach is much more effective. Over all experiments we have 9 functionally equivalent instances and our approach was able to detect 4 of them while using
partition-effects equivalence could detect 3.

6.4 RQ2: Restricting input space

**RQ2.** How robust is our approach in restricting the input space capturing only the inputs that expose the difference between versions?

For the Median kind, we had an increase for metric (diff %) of at least 0.25 in 70% of the instances and of at least 0.6 in 25% of them. On average the increase was 0.38. Those numbers show that the models obtained by solving the partition-effects input constraint are much more likely to trigger a divergence.

Similarly, for the Smallest kind, we had an increase for metric (diff %) of at least 0.35 in 30% of the instances. On average the increase was 0.20. Although not as significant as for the Median kind, those number show that the models obtained by solving the partition-effects input constraint are still more likely to trigger a divergence.

We can see a significant improvement in narrowing down the input space to a partition of it with high accuracy for exposing the behavioral difference between the versions. Therefore, these experiments support the claim that our approach provides a more precise specification for the input partition inducing behavioral divergence in terms of effects.
Chapter 7

Discussion

7.1 Threats to Validity

In this section, we discuss potential threats to the validity of our study that could affect the reliability and generalizability of our findings.

Dependence on Summary Format and Construction: A significant threat to the validity of our study arises from the reliance on the format and construction of method summaries. The accuracy, consistency, and comprehensiveness of these summaries are pivotal to the success of our approach. Although our approach is essentially agnostic to any symbolic execution tool, it is highly dependent on how the summaries were built. The number of partition effects and how they are built can vary significantly the results and the execution time. In our study, we use symc JPF to build the summaries and results may be overfit to it.

Dependency on Code Abstraction Selection: Another potential threat pertains to the selection of code to be abstracted within our approach. Variability in abstraction choices can lead to different results and conclusions. Moreover, the lack of standardized guidelines or automated procedures for abstraction may introduce subjectivity, potentially impacting the repeatability and consistency of our approach across different contexts.
Incomplete Summaries, Particularly with Loops: Incomplete method summaries, especially those involving loops and complex control flow structures, represent a notable threat to the validity of our findings. When a summary does not cover part of the input space is impossible to fully characterize the difference between methods. Our evaluation did not contain any loops or recurrence which may threaten the generalizability of our findings.

Simplicity of Evaluation Dataset: Our evaluation was conducted using a relatively simple dataset. This dataset’s simplicity might not adequately represent the diversity and complexity of real-world software applications. As a result, our findings may not fully capture the challenges and intricacies that practitioners encounter when applying our approach to more complex software systems. This limitation may affect the external validity of our study, making it challenging to extrapolate our results to larger, more intricate software projects.

To address these threats, future research should aim to standardize summary creation processes, investigate automated abstraction techniques, and expand the evaluation to more realistic and complex software scenarios. By addressing these threats, we can enhance the reliability and applicability of our approach in practical software analysis contexts.

7.2 Limitations

This study presents several limitations that should be acknowledged to provide a comprehensive understanding of the scope and applicability of our approach.

Limited Variety of Types: One notable limitation of our prototype is its restricted support for a variety of data types. Our current implementation primarily focuses on symbolic execution of integer and real data types. This limitation may hinder its ability to handle more diverse and complex data types commonly encountered in real-world software applications, potentially reducing its utility for broader software analysis tasks.
Manual Abstraction Process: The abstraction of code in our approach is currently a manual process and the criteria for determining what to abstract and to what level of detail were not exhaustively researched. This limitation introduces subjectivity into the abstraction process, as it relies on human judgment and expertise. As a result, the effectiveness and consistency of the abstraction process may vary across different users and scenarios. Future research should explore automated abstraction techniques and refine the guidelines for code abstraction to enhance the approach’s objectivity and scalability.

Incomplete Summaries: The accuracy and reliability of our approach heavily depend on the completeness of the method summaries used during analysis. In cases where summaries are incomplete, particularly with respect to loops or other complex program structures, some of the guarantees provided by our approach may be compromised. This limitation underscores the importance of comprehensive and accurate method summaries for achieving reliable results. Future work should focus on methods to address and mitigate the impact of incomplete summaries on the overall analysis process.

While our approach holds promise for advancing symbolic execution techniques in software analysis, these limitations should be considered when applying the approach to real-world scenarios. Addressing these limitations and conducting further research in these areas will be essential for enhancing the robustness and effectiveness of our approach in practical software analysis tasks.
Chapter 8

Conclusion

This thesis has introduced a novel technique for analyzing and characterizing the differences between two versions of a method. Building upon the foundations of Differential Symbolic Execution, our approach extends the concept of equivalence to effectively prove method functional equivalence or, when not equivalent, provide input constraints that narrow down the input space leading to divergent behaviors.

The development of a prototype tool implementing our approach allowed us to evaluate its effectiveness. Through a study using the IntroClass benchmark for Java, we sought to answer important research questions regarding its capability in classifying functionally equivalent programs.

Our findings demonstrate some of the potential of our approach in characterizing the difference between method versions. In particular, it showed to be valuable in constraining the input space representing more precisely the set of inputs that trigger a divergence. While our technique can establish program equivalence, we acknowledge that further evaluation and real-world application are needed to fully validate its practical competency.

This thesis has contributed a promising methodology for analyzing and characterizing method differences, offering a valuable technique for software engineers and researchers working in the domain of program analysis and verification. Our future work will develop an automated technique to abstract common code and generate the respective summaries, implement a robust tool including more complex datatypes for supporting client applications, and investigate the cost and ef-
fectiveness of this technique in automating software maintenance tasks.
Bibliography


Appendix A

Equivalence Proof

A.1 Partitio-Effects Equivalence implies Algorithmic Equivalence

Lemma A.1.1. If \( pe = (i, e) \) and \( pe' = (i', e') \) are equivalent partition-effects pair then \( i \land \neg i' \land e \) is unsat.

Proof. If \( pe \) and \( pe' \) are equivalent then we have the following implications

\[
i \land e \implies i' \land e'
\]

\[
i \land e \land \neg(i' \land e') \text{ is unsat}
\]

\[
i \land e \land (\neg i' \lor \neg e') \text{ is unsat}
\]

\[
(i \land e \land \neg i') \lor (i \land e \land \neg e') \text{ is unsat}
\]

\[
(i \land e \land \neg i') \text{ is unsat}
\]

\[\square\]

Theorem A.1.2. Let \( s \) and \( s' \) be two symbolic summaries, then if \( s \) and \( s' \) are partition-effects equivalent then they are algorithmically equivalent.
Proof. Let \( s = \{ (i_1, e_1), \ldots, (i_k, e_k) \} \) and \( s' = \{ (i'_1, e'_1), \ldots, (i'_{k'}, e'_{k'}) \} \) be two partition-effects equivalent summaries. Then there exist a function \( \sigma : s \mapsto s' \) such that for all \( (u, v) \equiv \sigma(u, v) \), where we say \( (r, s) \equiv (r', s') \) if, and only if, \( r \land s \iff r' \land s' \) for two partition-effects pairs \( (r, s) \) and \( (r', s') \). The input constraints of the elements of a method summary are disjoint, so there cannot be two elements in \( s' \) equivalent between themselves, additionally the summaries are partition-effects equivalent, thus, for each element \( p \) in \( s \) there exists a unique partition-effects pair \( p' \) in \( s' \) such that \( p \equiv p' \), therefore, the function \( \sigma \) is well defined. Looking at the dual \( \sigma' : s' \mapsto s \) one can see that

\[
\sigma(\sigma'(p')) \equiv \sigma'(p') \equiv p' \text{ for all } p' \in s'
\]

\[
\sigma'(\sigma(p)) \equiv \sigma(p) \equiv p \text{ for all } p \in s
\]

Since the input contraints of the elements of a method summary are disjoint we have that:

\[
\sigma(\sigma'(p')) = p' \text{ for all } p' \in s'
\]

\[
\sigma'(\sigma(p)) = p \text{ for all } p \in s
\]

therefore implying that \( \sigma \) is a bijection.

Consider the sequences of sets \( \{ s_n \}_{0 \leq n \leq k} \) and \( \{ s'_n \}_{0 \leq n \leq k} \) defined as below:

\[
s_0 = s
\]

\[
s_n = s_{n-1} \setminus \{ (i_n, e_n) \}
\]

\[
s'_0 = s'
\]

\[
s'_n = s'_{n-1} \setminus \{ \sigma(i_n, e_n) \}
\]

for \( 1 \leq n \leq k \). Using induction we can prove that for all \( 1 \leq p \leq k \) we have that \( \sigma((i_p, e_p)) \) belonging to \( s'_{p-1} \). The base case is trivial and the inductive step holds because \( \sigma \) is bijective.
Let’s prove that $\tau((i_p, e_p), s_{p-1}, \sigma(i_p, e_p), s'_{p-1}) = (s_p, s'_p)$ for all $1 \leq p \leq k$. For a given $1 \leq p \leq k$ let $(r, s) = \sigma(i_p, e_p)$. We know that

- $(i_p \land e_p) \implies (r \land s)$
- $(r, s) = \sigma((i_p, e_p)) \in s'_{p-1}$
- $(i_p, e_p) \in s_{p-1}$, by construction
- $i_p \land e_p \land \neg r$ is unsat, by Lemma A.1.1 for $\sigma((i_p, e_p))$ and $(i_p, e_p)$

Therefore applying the elimination rule we get

$$\tau((i_p, e_p), s_{p-1}, \sigma(i_p, e_p), s'_{p-1}) = (s_{p-1} \setminus \{(i_p, e_p)\}, s'_{p-1} \setminus \{\sigma(i_p, e_p)\})$$

$$= (s_p, s'_p)$$

Given that $\sigma$ is a bijection we conclude that $s_0$ and $s'_0$ have the same number of elements therefore $(s_k, s'_k) = (\emptyset, \emptyset)$ concluding that $s$ and $s'$ are algorithmically equivalent. \qed

### A.2 Algorithmic Equivalence implies Functional Equivalence

**Lemma A.2.1.** Let $p = (i, e)$ and $p' = (i', e')$ be two partition-effects pairs of a same method $m$. Then if $i \land e \implies i' \land e'$ and $i' \land e' \implies i$ then $i' \land e' \implies e$ and $p$ and $p'$ are equivalent.

**Proof.** It is enough to prove $i' \land e' \implies e$ since that combined with $i' \land e' \implies i$ entail the equivalence between $p$ and $p'$. Furthermore, since $i' \land e' \implies i$, it suffices to prove that $i \land e' \implies e$.

Let $\text{Write}(m) = \{g_1, \ldots, g_n\}$ be the set of written variables of method $m$. Then there exists symbolic expressions $f_1, \ldots, f_n$ and $f'_1, \ldots, f'_n$ in terms of the symbolic input variables such that the following holds:

$$e' = \bigwedge_{j=1}^n (g_j = f'_j) \quad \text{and} \quad e = \bigwedge_{j=1}^n (g_j = f_j) \quad (A.1)$$
Since \( i \land e \implies i' \land e' \) we have \( i \land e \implies e' \), thus, \( i \land e \implies e' \land e \) replacing the values of \( e \) and \( e' \) using Equation A.1 we arrive to:

\[
i \land \left( \bigwedge_{j=1}^{n} (g_j = f_j) \right) \implies \left( \bigwedge_{j=1}^{n} (g_j = f'_j) \right) \land \left( \bigwedge_{j=1}^{n} (g_j = f_j) \right)
\]

\[
\implies \left( \bigwedge_{j=1}^{n} (g_j = f'_j) \land (g_j = f_j) \right)
\]

\[
\implies \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right)
\]

Therefore \( i \land \left( \bigwedge_{j=1}^{n} (g_j = f_j) \right) \land \neg \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \) is unsat.

If \( i \land \neg \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \) were satisfiable one could get the model where \( (g_j = f_j) \) for all \( 1 \leq j \leq n \), since all variables \( g_j \) for \( 1 \leq j \leq n \) appear exactly once in the formula \( i \land \left( \bigwedge_{j=1}^{n} (g_j = f_j) \right) \land \neg \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \), hence, the formula \( i \land \left( \bigwedge_{j=1}^{n} (g_j = f_j) \right) \land \neg \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \) would be satisfiable which is not the case. Therefore, \( i \land \neg \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \) must be unsat implying that

\[
i \implies \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \quad \text{(A.2)}
\]

Combining the fact that \( i \land e' \implies e' \) with Equation A.2 we get:
\begin{align*}
i \land e' & \implies e' \land \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \\
& = \left( \bigwedge_{j=1}^{n} (g_j = f'_j) \right) \land \left( \bigwedge_{j=1}^{n} (f'_j = f_j) \right) \\
& = \left( \bigwedge_{j=1}^{n} (g_j = f'_j) \right) \land \left( f'_j = f_j \right) \\
& \implies \left( \bigwedge_{j=1}^{n} (g_j = f'_j) \right) \\
& = e
\end{align*}

Thus \( i \land e' \implies e \) concluding the proof. \hfill \Box

**Lemma A.2.2.** Let \( r, s, r', s' \) be summaries and \( p \in r, q \in s \) be partition-effects pairs such that \( \tau(p, r, q, s) = (r', s') \). Then we have that

\[
\left( \bigvee_{(i,e) \in r'} i \land e \right) \iff \left( \bigvee_{(i',e') \in s'} i' \land e' \right) \implies \left( \bigvee_{(i,e) \in r} i \land e \right) \iff \left( \bigvee_{(i',e') \in s} i' \land e' \right)
\]

**Proof.** Let \( p = (u, v) \) and \( q = (u', v') \). Then we have two cases to consider depending whether \( u' \land \neg u \land v' \) is unsat or not. Before getting into the cases, let us define \( t(h) := \left( \bigvee_{(i,e) \in h} i \land e \right) \) for a given summary \( h \). Therefore we must prove that

\[
\left( t(r') \iff t(s') \right) \implies \left( t(r) \iff t(s) \right) \tag{A.3}
\]

**Case 1.** \( u' \land \neg u \land v' \) is unsat

In this case, following the Elimination Rule (Rule 1), we have that \( r' = r \setminus \{p\} \) and \( s' = s \setminus \{q\} \). Then we conclude that
\[ t(r) = (u \land v) \lor t(r') \]
\[ t(s) = (u' \land v') \lor t(s') \]

Substituting those values in Equation A.3 we are required to prove that:

\[ \left( t(r') \iff t(s') \right) \implies \left( ((u \land v) \lor t(r')) \iff ((u' \land v') \lor t(s')) \right) \]

Additionally, since \( u' \land \neg u \land v' \) is unsat we have that \( u' \land v' \implies u \), using Lemma A.2.1, we have that \( p \) and \( q \) are equivalent. Hence, Equation A.3 holds.

**Case 2.** \( u' \land \neg u \land v' \) is sat

In this case, following the Elimination Rule (Rule 1), we have that \( r' = r \setminus \{p\} \) and \( s' = (s \setminus \{q\}) \cup \{(u' \land \neg u, v')\} \). We have that

\[ t(s) \lor (u' \land \neg u \land v') = (u' \land v') \lor t(s') = (u' \land v' \land u) \lor (u' \land v' \land \neg u) \lor t(s') \]

Therefore we can conclude that

\[ t(r) = (u \land v) \lor t(r') \]
\[ t(s) = (u' \land v' \land u) \lor t(s') \]

Substituting those values in Equation A.3 we are required to prove that:

\[ \left( t(r') \iff t(s') \right) \implies \left( ((u \land v) \lor t(r')) \iff ((u' \land v' \land u) \lor t(s')) \right) \]

Since we have that \( u \land v \implies u' \land v' \) and \( u \land v \implies u \) we get that \( u \land v \implies (u' \land u) \land v' \). In addition, \( (u' \land u) \land v' \implies u \), thus, using Lemma A.2.1 we get

55
that \( ((u' \land u), v') \) and \( (u, v) \) are equivalent, hence, \( (u \land v) \iff (u' \land v' \land u) \).

Therefore, Equation A.3 holds.

\[ \square \]

**Theorem A.2.3.** Let \( s \) and \( s' \) be two symbolic summaries, if \( s \) and \( s' \) are algorithmically equivalent then they are functionally equivalent.

**Proof.** Assume \( s \) and \( s' \) are algorithmically equivalent. Then, there exists a finite sequence \( \{ (r_i, s_i) \}_{0 \leq i \leq n} \) for some natural number \( n \) such that \( (s, s') = (r_0, s_0) \), \( (r_n, s_n) = (\emptyset, \emptyset) \) and for all \( 0 \leq i < n \) there exists \( p_i \in r_i \) and \( q_i \in s_i \) such that we have either \( \tau(p_i, r_i, q_i, s_i) = (r_{i+1}, s_{i+1}) \) or \( \tau(q_i, s_i, p_i, r_i) = (r_{i+1}, s_{i+1}) \).

Similarly to previous proof let us define \( \iota(h) := (\lor_{(i, e) \in h} i \land e) \) for a given summary \( h \).

Therefore, it is sufficient to prove that \( \iota(s) \iff \iota(s') \). We will prove that by proving that for all \( 0 \leq i \leq n - 1 \), \( t(r_i) \iff t(s_i) \). The proof will follow by induction on the value \( n - i \).

**Base Case.** \( n-i = 1 \), that is \( i = n-1 \).

Since the Elimination Rule removes at most one element from each summary at each step and \( (r_n, s_n) = (\emptyset, \emptyset) \) we can conclude that \( r_{n-1} = \{ (u, v) \} \) and \( s_{n-1} = \{ (u', v') \} \) are unitary sets and that both become empty after applying the Elimination Rule. WLOG let us assume that \( \tau((u, v), r_{n-1}, (u', v'), s_{n-1}) = (r_n, s_n) = (\emptyset, \emptyset) \) (The other case is completely analogous). Then we have that:

\[
(u \land v) \implies (u' \land v') \quad \text{and} \quad u' \land -u \land v' \text{ is unsat}
\]

Since \( u' \land -u \land v' \) is unsat we get \( u' \land v' \implies u \) and using Lemma A.2.1 we arrive to

\[
\iota(r_{n-1}) = (u \land v) \iff (u' \land v') = \iota(s_{n-1})
\]

Concluding the base case.
**Inductive Step.** Let $n - i = k$, for some $1 \leq k \leq n - 1$, and that $\iota(r_{n-k}) \iff \iota(s_{n-k})$, we will prove that $\iota(r_{n-k-1}) \iff \iota(s_{n-k-1})$.

Since the sequence $\{(r_i, s_i)\}_{0 \leq i \leq n}$ is such that for all $0 \leq i < n$ there exists $p_i \in r_i$ and $q_i \in s_i$ such that either $\tau(p_i, r_i, q_i, s_i) = (r_{i+1}, s_{i+1})$ or $\tau(q_i, s_i, p_i, r_i) = (r_{i+1}, s_{i+1})$, then there exists $p_{n-k-1} \in r_{n-k-1}$ and $q_{n-k-1} \in s_{n-k-1}$ such that either $\tau(p_{n-k-1}, r_{n-k-1}, q_{n-k-1}, s_{n-k-1}) = (r_{n-k}, s_{n-k})$ or $\tau(q_{n-k-1}, s_{n-k-1}, p_{n-k-1}, r_{n-k-1}) = (r_{n-k}, s_{n-k})$. In both cases we can apply Lemma A.3 and arrive to

$$\iota(r_{n-k}) \iff \iota(s_{n-k}) \iff \iota(r_{n-k-1}) \iff \iota(s_{n-k-1})$$

Using the induction hypothesis, $\iota(r_{n-k}) \iff \iota(s_{n-k})$, we get that $\iota(r_{n-k-1}) \iff \iota(s_{n-k-1})$ concluding the Inductive Step.

Since $\iota(r_i) \iff \iota(s_i)$ for all $0 \leq i \leq n - 1$, in particular

$$\iota(s) = \iota(r_0) \iff \iota(s_0) = \iota(s')$$

Thus concluding the proof. \qed
Appendix B

Code Example

```java
package eval.median.median;

public class Median {
    public int exec(int arg1, int arg2, int arg3) throws Exception {
        int num1=0, num2=0, num3=0, median=0;
        int bigger12=0, smaller12=0;
        int output=0;
        num1=arg1;
        num2=arg2;
        num3=arg3;
        if(num1 < num2){
            bigger12=num2;
            smaller12=num1;
        } else {
            bigger12=num1;
            smaller12=num2;
        }
        if(bigger12 < num3){
            median=bigger12;
        } else if(num3 > smaller12){
            median=num3;
        } else {
            median=smaller12;
        }
        output +=(( median ));
        return output;
    }
}
```

Figure B.1: Code snippet automatically generated from the benchmark code of Figure B.2.
package introclassJava;

class IntObj { public int value; public IntObj() { value = i; }}
class FloatObj { public float value; public FloatObj() { value = i; }}
class LongObj { public long value; public LongObj() { value = i; }}
class DoubleObj { public double value; public DoubleObj() { value = i; }}
class CharObj { public char value; public CharObj() { value = i; }}

public class Median {
    public java.util.Scanner scanner;
    public String output = " ";

    public static void main(String[] args) throws Exception {
        Median mainClass = new Median();
        String output;
        if (args.length > 0) {
            mainClass.scanner = new java.util.Scanner(args[0]);
        } else {
            mainClass.scanner = new java.util.Scanner(System.in);
        }
        mainClass.exec();
        System.out.println(mainClass.output);
    }

    public void exec() throws Exception {
        IntObj num1 = new IntObj(), num2 = new IntObj(), num3 = new IntObj(), median = new IntObj();
        IntObj bigger12 = new IntObj(), smaller12 = new IntObj();
        output += (String.format("Please enter 3 numbers separated by spaces > "));
        num1.value = scanner.nextInt();
        num2.value = scanner.nextInt();
        num3.value = scanner.nextInt();
        if (num1.value < num2.value) {
            bigger12.value = num2.value;
            smaller12.value = num1.value;
        } else {
            bigger12.value = num1.value;
            smaller12.value = num2.value;
        }
        if (bigger12.value < num3.value) {
            median.value = bigger12.value;
        } else if (num3.value > smaller12.value) {
            median.value = num3.value;
        } else {
            median.value = smaller12.value;
        }
        output += (String.format("%d is the median
", median.value));
        if (true) return;
    }
}

Figure B.2: Original code example from IntroClass benchmark for Java.