# OBTAINING REAL-WORLD BENCHMARK PROGRAMS FROM OPEN-SOURCE REPOSITORIES THROUGH ABSTRACT-SEMANTICS PRESERVING TRANSFORMATIONS

by

Maria Anne Rachel Paquin

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

## DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Maria Anne Rachel Paquin

Thesis Title: Obtaining Real-World Benchmark Programs From Open-Source Repositories Through Abstract-Semantics Preserving Transformations

Date of Final Oral Examination: 4th May 2020

The following individuals read and discussed the thesis submitted by student Maria Anne Rachel Paquin, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

| | |
|---|---|
| Elena Sherman, Ph.D. | Chair, Supervisory Committee |
| Catherine Olschanowsky, Ph.D. | Member, Supervisory Committee |
| Maria Soledad Pera, Ph.D. | Member, Supervisory Committee |

The final reading approval of the thesis was granted by Elena Sherman, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

# ACKNOWLEDGMENTS

I would like to begin by thanking my advisor, Dr. Elena Sherman, for all of her guidance and support throughout this research. Over the last two years she not only taught me what it means to be a researcher, but instilled in me the confidence to be one myself. It is because of her faith in my ability and the countless hours she spent sharing her advice, knowledge and ideas that I was able to finish this thesis, and for that, I am eternally grateful.

I would like to thank Dr. Sole Pera and Dr. Catherine Olschanowsky for serving as my committee members and providing insightful comments and suggestions during my proposal. Thanks to Dr. Amit Jain for guiding me through my academic career and helping me transition from academia to industry.

I am indebted to my friends and family, for providing both financial and emotional support and for helping me endure the rigors of grad school. I would not have been able to do it without them. Finally, I owe a deep and heart-felt thank you to Jim Pelton. His words of encouragement and advice gave me the strength to persevere during the most difficult times.

# ABSTRACT

Benchmark programs are an integral part of program analysis research. Researchers use benchmark programs to evaluate existing techniques and test the feasibility of new approaches. The larger and more realistic the set of benchmarks, the more confident a researcher can be about the correctness and reproducibility of their results. However, obtaining an adequate set of benchmark programs has been a long-standing challenge in the program analysis community.

In this thesis, we present the APT tool, a framework we designed and implemented to automate the generation of realistic benchmark programs suitable for program analysis evaluations. Our tool targets intra-procedural analyses that operate on an integer domain, specifically symbolic execution. The framework is composed of three main stages. In the first stage, the tool extracts potential benchmark programs from open-source repositories suitable for symbolic execution. In the second stage, the tool transforms the extracted programs into compilable, stand-alone benchmarks by removing external dependencies and nonlinear expressions. In the third stage, the benchmarks are verified and made available for the user.

We have designed our transformation algorithms to remove program dependencies and nonlinear expressions while preserving their semantics-equivalence in the abstraction of symbolic analysis. That is, we want the information the analysis computes on the original program and its transformed version to be equivalent. Our work provides static analysis researchers with concise, compilable benchmark programs that are relevant to symbolic execution, allowing them to focus their efforts on advancing analysis techniques. Furthermore, our work benefits the software engineering community by enabling static analysis researchers to perform benchmarking with a large, realistic set of programs, thus strengthening the empirical evidence of the advancements in static program analysis.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

**APT** – Abstract-Semantics Preserving Transformations

**AST** – Abstract Syntax Tree

**SE** – Symbolic Execution

**PC** – Path Constraint

**SPF** – Symbolic PathFinder

**SMT** – Satisfiability Modulo Theories

**LV** – Live Variable

**RD** – Reaching Definitions

# LIST OF SYMBOLS

$s$: a statement

$S_\star$: a sequence of statements

$init(S_\star)$: the entry statement in a sequence of statements

$\{final(S_\star)\}$: the exit statement(s) in a sequence of statements

$L$: a domain of data-flow values

$\sqcup$: the meet operator

$\iota$: a data-flow value for initialization

$f_s$: a transfer function for the statement $s$

$\sigma$: a concrete program state

$\widehat{\sigma}$: an abstract program state

$\top$: the top element in $L$

$\bot$: the bottom element in $L$

$U$: the universal set

$A$: an analysis

$entry(s)$: the data-flow value before executing statement $s$

$exit(s)$: the data-flow value after executing statement $s$

$gen_s$: the set of definitions generated by a statement $s$

$kill_s$: the set of definitions killed by a statement $s$

$FV(S_\star)$: the set of free variables in the sequence of statements $S_\star$

$V$: the set of integer variables in a program

$\mapsto$: notates a mapping from a program variable to a value, e.g., $x \mapsto 1$

$\langle s, \sigma \rangle$: a configuration of semantics

$\to$: a transition relation between configurations of semantics

$\xrightarrow{*}$ : a multi-step reduction transition relation

$\Gamma$: the local type system, i.e., a typing function which determines the types of variables

$Img(\Gamma)$: the image of the typing function $\Gamma$, i.e., the set of types that $\Gamma$ may output

$\tau$: a data type

$\mathcal{S}^\tau$: the semantic function for type $\tau$

$e^\tau$: an expression of type $\tau$

$D_A$: an abstract domain of $A$

$\mathcal{F}_A$: the transfer functions of $A$

$\mathcal{D}_A$: the set of possible values to which $A$'s abstract semantics functions $\widehat{\mathcal{S}}^\tau_A$ can evaluate an expression of type $\tau$

$P$: a Java class file

$T_1$: the set of types of the project $P$ resides in

$T_P$: the set of types defined in $P$

$\Gamma_2$: the local type system for SPF

$T_2$: the set of types defined in SPF or the Java standard library, i.e., the set of types for which we would like P to be compilable

$Var(s)$: the variables either defined or used in the statement $s$

$FV(e)$: the set of free variables in the expression $e$

$sym$: a symbolic variable

$op$: an infix operator

# CHAPTER 1

# INTRODUCTION

Static program analysis is a technique used to reason about a program's behavior without actually executing the program. These types of analyses have a range of applications such as defect detection, program verification and compiler optimization [20]. Because static analyses are routinely used to ensure the quality and correctness of software systems, researchers are continually developing new techniques and optimizing existing ones. The ability of these techniques to handle the complexity of real-world code relies on the soundness of their testing. Thus, researchers emphasize the importance of evaluating program analysis techniques with benchmark programs that are representative of real-world applications. The larger and more realistic the set of benchmark programs, the more confidence the researcher can have in the evaluation of the analysis technique. However, various analyses have different requirements of their benchmarks. While light-weight analyses operate on the text of a program or at most its abstract syntax tree (AST) structure, more involved analyses such as data-flow analysis require compilable programs. Furthermore, heavy-weight program analyses are designed to operate on programs with certain properties or features. For example, symbolic execution, a path-sensitive heavy-weight analysis technique, may focus on integer or string types, or operate on an intra-procedural level. Consequently, tools that implement symbolic execution may be limited in the programs they can effectively analyze. As a result, researchers have difficulty finding programs suitable for evaluating heavy-weight analyses.

The objective of this work is to help researchers with this task. We do so by developing a framework that automatically identifies, downloads and transforms programs from open-source repositories to create a suite of benchmark programs which can be used to analyze the symbolic execution tool Symbolic PathFinder. We implement an instance of this framework to obtain 902 method benchmarks from 611 classes, increasing by nine times the number of available benchmark methods for this tool. Moreover, we show the impact of increased numbered of benchmarks on the results of a previous study.

## 1.1 Scarcity of Benchmarks for Heavy-Weight Analyses

To demonstrate the difficulty finding programs suitable for evaluating heavy-weight analyses, the authors of [15] conduct a literature review of benchmark programs classified by static analysis complexity. The levels of complexity are divided into light, medium and heavy, and four papers of each complexity type are sampled from recent software engineering and programming languages conferences. For each paper, the survey considers the number and source of the benchmarks used in the study, as well as program characteristics which are used to help approximate program size.

The results of the review are shown in Table 1.1. The Feature Type column describes the program features each study was analyzing. Since medium and heavy-weight analyses focus on a program behavior, entries in this column are simply the number of lines of code or call graph edges. The last column contains the total number of features analyzed in each study.

The data shows a substantial difference between the number of programs used in light-weight analyses compared to medium and heavy-weight. While tens (or hundreds) of thousands of programs are used in light-weight analyses, the number of programs available for medium to heavy-weight is several orders of magnitude smaller. Furthermore, using the total number of programs and features analyzed, we can approximate the program size in each type of analysis. Even though medium and heavy-weight analyses use a similar number of programs, the programs used in the heavy-weight analyses are substantially smaller.

Without an adequate set of benchmark programs, researchers are left to manually find and adapt real-world programs for their specific analyses. Consequently, they focus time and energy on generating benchmarks that could be spent advancing techniques. In addition, manual code transformations are error prone and hinder scalability of the experiment. In this thesis, we address the problem of obtaining a large set of benchmark programs, specifically for symbolic execution. To demonstrate the challenges of obtaining realistic benchmark programs suitable for symbolic execution, we first provide an overview of the technique.

## 1.2 Symbolic Execution

Symbolic execution [2] is a static analysis technique used to systematically explore multiple program paths. In contrast to normal execution, symbolic execution does not require specific input and is able to reason about different paths of executions depending on potential input. For this reason, symbolic execution has been shown to be a useful technique for defect detection [23] and test case generation [10].

During symbolic execution, input values are represented as symbolic variables

| SA Type | Conf & Year | # Programs | Feature Type | Total #Features |
|---|---|---:|---|---:|
| Light-weight | ICSE’14 [14] | 31,432 | Java files | 9,093,216 |
| | ICPC’15 [42] | 16,221 | License changes commits | 1,731,828 |
| | MSR’16 [27] | 554,864 | Methods with `catch` blocks | 10,862,172 |
| | MSR’16 [37] | 28,466 | Commit logs | 20,130,474 |
| Medium-weight | PLDI’17 [29] | 14 | Lines of code | 293,154 |
| | POPL’17 [46] | 25 | Call graph edges | 1,097,676 |
| | PLDI’17 [41] | 7 | | 445,500 |
| | PLDI’17 [39] | 12 | | 845,489 |
| Heavy-weight | ISSTA’15 [22] | 7 | Lines of code | 5,747 |
| | ICSE’15 [36] | 10 | | 3,590 |
| | ICSE’16 [11] | 7 | | 10,700 |
| | ASE’16 [24] | 16 | | 2,386 |

Table 1.1: Static analysis complexity types and evaluation scope.

rather than concrete values. As execution proceeds, it constructs a logical formula over the symbolic input variables, called path constraints (PC). The solutions describe the branch conditions satisfied to reach the program state of each explored path. Upon reaching a conditional branch, the symbolic execution engine generates two PCs, each extended from the original PC by conjoining the constraints of all previously taken branches with the current branch constraint. The symbolic execution engine proceeds independently along both branches, and in this way all possible program paths are explored simultaneously. The feasibility of a new PC is checked by a constraint solver. Moreover, the solution to the PC yields concrete values of input that cause the program to follow that particular path, which can then be used for testing.

### 1.2.1 Example

Consider the code listed in Figure 1.1 [34], which switches the values of $x$ and $y$. During concrete, i.e., normal execution, specific values of input are used for $x$ and $y$ and only one program path is explored. For example, if $x = 1$ and $y = 2$, the conditional statement on line 2 evaluates to false, the false branch is followed and the execution is finished.

In contrast, during symbolic execution $x$ and $y$ are represented by the symbolic values $X$ and $Y$, respectively. A symbolic execution engine maintains for each program state: 1) a map of program variables to their symbolic values and 2) a PC encoding the branches taken to reach that program state.

The symbolic execution of the example code can be represented as a tree, as shown in Figure 1.1. Each program state is represented as a node in the tree. Initially, the PC is 'true' since there are no restrictions on $x$ and $y$. The variable $x$ maps to symbolic value $X$ and, similarly, $y$ maps to $Y$. Upon executing the conditional statement on line 2, both branches are explored, shown by the first branch of the tree. The right child node corresponds to following the 'false' branch, i.e., when $X \leq Y$, shown by the path constraint for this program state.

The left branch corresponds to the conditional statement on line 2 being true, i.e., $X > Y$. Upon executing line 3, $x = x + y$, a new program state is created with the symbolic variable $x$ now mapping to $X + Y$ (note the PC stays the same for non-branching instructions). After executing line 4, $y$ maps to $X$, and after line 5, $x$ maps to $Y$. When the next branch is encountered, i.e., the conditional statement on line 6, the symbolic execution engine creates two new program states, shown by the bottom two nodes in the symbolic execution tree. The left node corresponds to the 'true' branch, i.e., $(x - y) > 0$, and the right node corresponds to the false branch, $(x - y) \leq 0$. Note the path constraint for the program state branching on the false condition is unsatisfiable, since $Y - X > 0$ implies $Y > X$, and there are no values of $x$ and $y$ such that $x > y$ and $y > x$ are both true. Branches that are unsatisfiable, as in this case, do not need to be analyzed further by the symbolic execution engine.

| **Path** | **Path Constraint** | **Input** |
|---|---|---|
| 1,2,9 | $X \leq Y$ | $x = 1$, $y = 1$ |
| 1,2,3,4,5,6,8,9 | $X > Y \land Y - X \leq 0$ | $x = 2$, $y = 1$ |
| 1,2,3,4,5,6,7,9 | $X > Y \land Y - X > 0$ | none |

Table 1.2: Possible program input for paths explored.

A constraint solver can be used to solve the path constraints, generating concrete input values that cause each program path to execute. Users can then create a high-coverage test suite, or determine specific input values that cause a program to execute along a path with a known bug for manual testing. For example, setting both $x$ and $y$ equal to 1 causes the example code to execute along the path $1, 2, 9$. Table 1.2 shows concrete input values for $x$ and $y$ for each of the paths explored.

In this example, the constraints are in the theory of linear integer arithmetic, for which the decision problem is decidable. However, the constraints generated from real-world software systems can be far more complex. Certain types of constraints, e.g., those that involve non-linear arithmetic, can lead to undecidable problems and may not be solvable, depending on the theories supported by the specific constraint solver. If the constraints cannot be solved, symbolic execution cannot generate concrete input values. Thus, programs with complex path constraints cannot be used to test the current capabilities of the symbolic execution engine.

### 1.2.2 Symbolic PathFinder

There are many tools implementing symbolic execution, such as KLEE [9], SAGE [31] and CREST [8]. In this work we focus on Symbolic PathFinder (SPF), which is an extension project of Java PathFinder [44]. Java PathFinder is a software verification framework developed by the NASA Ames Research Center. At its core is a customized

```
int x,y;
if(x > y){
        x = x + y;
        y = x - y;
        x = x - y;
        if(x - y) > 0{
                assert false;
        }
}
```

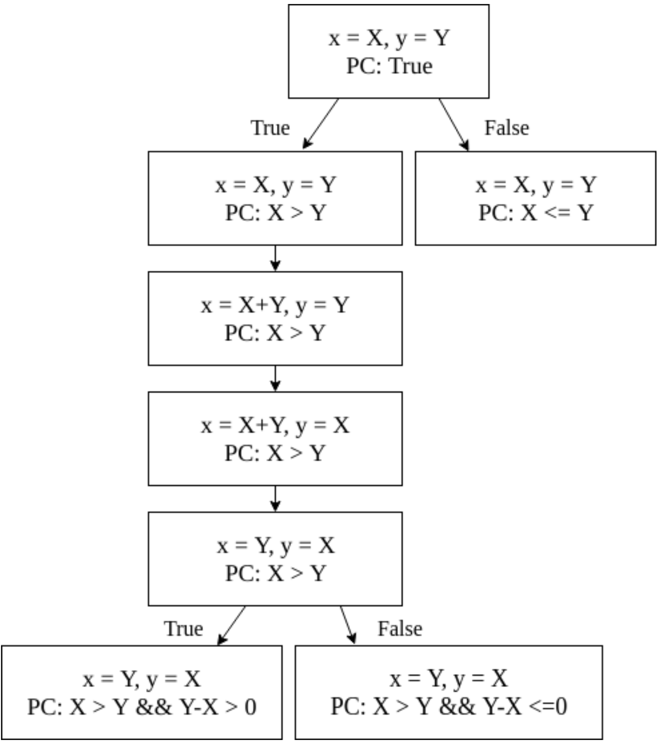


Figure 1.1: Example code with corresponding symbolic execution tree to illustrate symbolic execution.

Java Virtual Machine that allows for exploring different program paths, storing each explored program state and backtracking when it visits a previously stored state.

SPF is an extension of Java PathFinder which performs symbolic execution of Java bytecode; that is, programs can be executed with symbolic values for numeric and boolean input variables. SPF uses the model checking supported by Java PathFinder to backtrack and explore different paths of execution. The constraints that are created during symbolic execution are solved with off-the-shelf decision procedures, i.e., satisfiability modulo theories (SMT) solvers.

Although symbolic execution can exhaustively explore all possible paths of execution, in practice this is not likely to scale to large programs. Tools implementing symbolic execution face many challenges when it comes to the complexity of real-world code. Some of the key challenges are (i) state space explosion, i.e., the number of program states increases exponentially with program size, and may be infinite in the presence of loops, (ii) memory management, e.g., manipulating complex objects may give rise to addresses stored by symbolic expressions, (iii) interactions with the environment, e.g., system or library calls, and as previously discussed (iv) complex path constraints. Thus SPF is less effective at processing programs that are large and overly complex, contain rich object behavior, or perform library/system calls. In addition to these obstacles, SPF only performs intra-procedural analysis and operates on types supported by off-the-shelf constraint solvers. For these reasons, finding real-world programs that SPF can effectively analyze is a challenging task.

| Conf. &Year | Total | ASW | Appolo | Bin | OAE | MER | TSAFE | TCAS | TreeMap | WBS |
|---|---|---|---|---|---|---|---|---|---|---|
| TACAS'07 [1] | 2 | | | | | | | ✓ | ✓ | |
| ISSTA'08 [35] | 1 | | | | ✓ | | | | | |
| ISSTA'10 [38] | 5 | ✓ | | ✓ | | | | | ✓ | ✓ |
| ISSTA'11 [33] | 2 | | ✓ | | | | ✓ | | | |
| ISSTA'12 [45] | 5 | | ✓ | | | ✓ | | ✓ | | ✓ |
| FSE'12 [43] | 6 | | | ✓ | | | | ✓ | ✓ | |
| ICSE'13 [17] | 2 | | | | ✓ | | | | | |
| FSE'13 [6] | 1 | | | | | | ✓ | | | |
| FSE'14 [18] | 2 | | | | ✓ | ✓ | | | | |
| PLDI'14 [5] | 2 | | ✓ | | | | ✓ | | | |
| ISSTA'15 [22] | 7 | | | ✓ | | ✓ | | ✓ | ✓ | |
| ICSE'15 [36] | 10 | ✓ | ✓ | | | | | ✓ | | ✓ |
| ASE'16 [24] | 16 | | | | | | ✓ | ✓ | | |
| ICSE'16 [11] | 7 | | ✓ | | | | ✓ | | ✓ | |

Table 1.3: Commonly used artifacts for evaluating the Symbolic PathFinder (SPF) tool.

## 1.3 Scarcity of Benchmarks for Symbolic PathFinder

The authors of [15] support our claim that finding programs suitable for SPF is challenging. They examined 14 papers in software engineering and programming languages conferences over the last decade and considered the benchmark programs used in each evaluation. Results of this study are shown in Table 1.3. The first two columns show the publication and the total number of benchmarks used in the study. The remaining columns list commonly used benchmark programs among the 14 studies. A checkmark ✓indicates that the authors used a corresponding benchmark in their analysis. The total number of programs used as well as the commonality of programs among 14 publications demonstrate the scarcity of programs available for analysis with SPF.

## 1.4 Thesis Statement

*Automating the selection and adaption of benchmark programs for SPF will advance symbolic execution research by providing a large number of realistic programs to evaluate new and existing symbolic execution techniques.*

To support this statement, we answer the following research questions:

1. Can we use source code from open-source repositories to obtain a suite of compilable benchmark programs tailored specifically to SPF?
2. Can a larger set of benchmark programs affect existing evaluations of symbolic execution techniques?

## 1.5 Contributions

The contributions of this thesis are the following:

1. Design and implementation of a framework that identifies, downloads and transforms programs into benchmarks suitable for SPF.
2. Design of abstract-semantics preserving algorithms that eliminate a program's external dependencies and replace its nonlinear expressions.
3. Empirical evidence that demonstrates the necessity of an automated tool which generates realistic benchmark programs for SPF.

Next, we provide an overview of existing benchmark sources and explain why each is insufficient for SPF. We then provide sufficient background information on data-flow analysis and semantic program equivalence to explain our framework and corresponding implementation. We address the research questions in the evaluation of our tool, and conclude with a summary of our contributions and ideas for future work.

# CHAPTER 2

# REVIEW OF AVAILABLE BENCHMARK SOURCES

This chapter presents and evaluates several existing sources researchers can use to obtain program benchmarks. As discussed in Chapter 1, researchers cannot choose symbolic execution benchmarks arbitrarily; the programs must contain methods that symbolic execution can process, and must produce non-trivial results, i.e., results useful to researchers. An ideal program would contain code with the following properties: (i) integer parameters to be executed symbolically, (ii) linear arithmetic operations over the integer parameters, (iii) conditional statements over integer expressions, (iv) a limited number of nested conditionals, and (v) a limited number of loops. The first three requirements ensure that complex, non-empty path constraints are generated when the program is executed symbolically. The last two requirements mitigate the problem of state space explosion. Recall that the number of paths can grow exponentially with the number of conditional statements, and similarly, new path constraints can be generated each time the loop condition is checked. In fact, if the upper bound of the loop condition is symbolic, path constraints will be generated infinitely unless a limit on the search depth is set.

Below are some examples of desirable code structures. The `compare` method in Listing 2.1 is from one of the commonly used programs identified in [15]. The other two methods, `abort` shown in Listing 2.2 and `myMethod` in Listing 2.3, are from example programs included with the SPF distribution.

```
private int compare(int k1, int k2) {
    if (k1 < k2) {
        return -1;
    } else if (k1 == k2) {
        return 0;
    } else {
        return 1;
    }
}
```

Listing 2.1: A method from TreeMap.

```
public void abort(int altitude, boolean controlMotorFired) {
    if (!controlMotorFired)
        failures.add(Failure.LAS_CNTRL);
    if (altitude <= 120000) {
        if (controlMotorFired) {
            setNextState("abortLowActiveLAS");
        } else {
            setNextState("abortPassiveLAS");
        }
    }
    if (altitude >= 120000) {
        setNextState("abortHighActiveLAS");
    }
}
```

Listing 2.2: A method from SPF's example program ExampleAbort.

```
public int myMethod(int x, int y) {
    int z = x + y;
    if (z > 0) {
        z = 1;
    } else {
        z = z - x;
    }
    if (x < 0) {
        z = z * 2;
    } else if (x < 10) {
        z = z + 2;
    } else {
        z = -z;
    }
    if (y < 5) {
        z = z - 12;
    } else {
        z = z - 30;
    }
    return z;
}
```

Listing 2.3: A method from SPF's example program MyClassOriginal.

## 2.1 Existing Benchmark Repositories

To address the lack of benchmarks available for empirical studies, researchers in the programming languages and software engineering communities have created several benchmark repositories such as SIR [12], DaCapo [4] and Qualitas Corpus [40].

Table 2.1: SIR benchmarks for symbolic execution tools.

| Program | LOC | Classes | Downloads |
|---:|---:|:---:|:---:|
| Array-Partition | 13 | 1 | 594 |
| Binary-Search-Tree | 130 | 4 | 671 |
| Doubly-Linked-List | 277 | 1 | 385 |
| Sorting | 13 | 1 | 468 |
| Vector | 254 | 1 | 320 |
| Binary-Heap | 72 | 2 | 425 |
| Disjoint-Set | 35 | 1 | 364 |
| Red-Black-Tree | 334 | 1 | 396 |
| Stack | 114 | 5 | 334 |

However, these repositories only contain between 14 and 112 outdated projects, and do not provide a sufficient number of programs suitable for symbolic execution.

### 2.1.1 SIR

The Software-artifact Infrastructure Repository (SIR) contains artifacts that researchers can use to experiment with software testing and regression testing techniques. The infrastructure offers all of the software-related artifacts to perform controlled experimentation, including multiple program versions, test suites and fault data. SIR includes 68 programs written in Java, most recently updated January 1, 2015. Nine programs are targeted for symbolic execution and test case generation tools such as SPF. These nine programs were uploaded July 14, 2011 and are shown in Table 2.1 with their size (in lines of code), class count, and number of times each one was downloaded from SIR. From this table, we can see that all of the programs are rather small, only five of which contain more than 100 lines of code. Furthermore, the number of downloads of each program demonstrates a community need for such a set of programs.

Table 2.2: DaCapo Benchmarks.

| Program | Description |
|---:|---|
| avrora | A simulation and analysis framework for AVR microcontroller. |
| batik | A toolkit for handling images in the Scalable Vector Graphics format. |
| eclipse | An integrated development environment. |
| fop | A print formatter. |
| h2 | A relational database management system. |
| jython | A Python interpreter. |
| luindex | A text indexing tool. |
| lusearch | A text search tool. |
| pmd | A source code analyzer. |
| sunflow | An image rendering system. |
| tomcat | A server executing Java Servlet and Java Server Pages |
| tradebeans | An application emulating an online stock trading system |
| tradesoap | An application emulating an online stock trading system |
| xalan | An XSLT processor for transforming XML documents. |

### 2.1.2 DaCapo

The DaCapo benchmarks are a set of freely available, general-purpose Java applications. The creators of DaCapo chose open-source programs that exhibited rich code complexity and demanding memory requirements. They also considered ease of use and testing, excluding GUI applications and targeting programs with minimal dependencies outside the host JVM. The suite consists of 14 benchmarks shown in Table 2.2.

Although these programs are much larger and more complex than those provided by SIR, the source code is not included in the benchmark suite. Instead, an 'execution harness' is provided which invokes the executable for each program with supplied input. This may be a problem for some researchers who wish to have the original source-code to reason about the correctness of their results.

### 2.1.3 Qualitas Corpus

The Qualitas Corpus is a curated collection of open-source Java programs. The programs are aggregated from a variety of sources and documented with metadata, with the objective of reducing the cost and increasing the reproducibility of empirical studies on code structure. The current release of the corpus includes 112 systems.

Although the corpus contains a larger set of programs than SIR and DaCapo (in fact, 6 of the 14 DaCapo benchmarks are included in the corpus), it is not consistently updated. Since its release in May 2013, it has only been updated once, in September 2013. Without being well-maintained, a benchmark repository cannot consistently provide researchers with programs that utilize modern programming practices.

## 2.2 Mining Open-Source Repositories for Benchmarks

Since benchmark repositories cannot consistently provide a large set of programs with modern complexity, researchers are left to manually find and adapt real-world programs. With software mining tools such as Boa [13] or RepoReaper [26], researchers can query open-source repositories to search for projects that contain programs specific to their needs.

### 2.2.1 RepoReaper

RepoReaper is a framework that allows researchers to select GitHub repositories that contain an engineered software project rather than a toy project such as a homework assignment. An implementation of the framework is presented as a software mining tool called *reaper*, which computes several meta-data for GitHub projects such as frequency of commits, ratio of test code to all code, whether or not a continuous integration service is used, etc. These attributes are then used to determine if the repository is an engineered project, using classifiers trained on a manually labeled dataset.

The software mining tool *reaper* is available as an open-source project. The project includes the dataset containing the raw values of 1,857,423 repositories, which can be downloaded as a CSV.

### 2.2.2 BOA

Boa is a domain-specific programming language and infrastructure for analyzing large-scale software repositories. The objective of Boa is to help scientists and researchers analyze the wealth of information contained in ultra-large-scale repositories by providing them the means to test their mining software repository hypotheses in a systematic and reproducible fashion.

### 2.2.3 Problem with Mining Open-Source Repositories

Mining open-source repositories may be a viable option for light-weight analyses which operate on the text of a program or at most its AST structure. However, the process of finding and adapting programs for medium to heavy-weight analyses is still a challenge. Since these types of analyses require compilable source code, the researcher would have to download and build the projects in order to prepare benchmarks for evaluation. Furthermore, the heavy-weight analysis that we target in this work, symbolic execution, performs intra-procedural analysis in an integer domain. Thus, the process of finding and preparing programs suitable for symbolic execution would involve a substantial amount of work and careful documentation.

## 2.3 Synthetic Programs

Another approach researchers might take is to synthetically generate programs according to some specifications. One such program generation tool is RUGRAT [21], which researchers have used to evaluate different software engineering tasks for Java programs, including program analysis, and test case generation techniques with high structural coverage of programs [19] [30].

The objective of RUGRAT is to generate a structurally diverse set of Java programs parameterized by the frequencies with which a particular language construct appears in generated programs. Users can specify a variety of program properties, such as number of class fields, maximum nested conditional statements and interface depths.

### 2.3.1 Problem with Synthetically Generated Programs

We investigated whether programs automatically generated with RUGRAT are adequate for training a machine learning model which determines SPF's configurations for a given Java method [**?**]. To do this, we compared the performance of a model trained on real programs with that of a model trained on synthetic programs. Our results indicated that using synthetic data alone to train a model may be insufficient, as such a model was unable to learn the relationship between attributes that was found by models trained using data generated from real programs. Thus in certain contexts, synthetically generated programs cannot fully represent real-world programs.

# CHAPTER 3

# BACKGROUND

Before presenting our framework for abstract-semantic preserving transformations, we provide the necessary background information on data-flow analysis and semantic program equivalence.

## 3.1 Data-Flow Analysis

One way compilers can improve the efficiency of code is by optimizing register allocation. Accessing a value stored in a register is faster than reading from memory or the disk, and since we have a limited number of registers, we would like to optimize the way they are allocated. For example, we do not need to store the value of a variable if it will be overwritten before it is used. To identify those scenarios, researchers developed Live Variable (LV) analysis that computes for each program point the set of variables that are live, i.e., variables whose values could be used during some execution of the program starting from that point.

Data-flow analysis frameworks define algorithms for iteratively gathering information about the flow of data along program execution paths and can be used to instantiate a specific analysis such as LV. The analyses compute invariants for each program state, disregarding the rest of the program details. In general, no analysis operates on a concrete representation of the state; it extracts and abstracts only the information relevant to the analysis.

### 3.1.1 Framework

Although there are many data-flow analysis frameworks, we use the following as defined in [28] for development of our APT tool:

- A data-flow graph of a sequence of statements $S_\star$
- A direction of flow, either forwards or backwards
- Extremal labels, either $init(S_\star)$ or $\{final(S_\star)\}$
- A semilattice with a domain of values $L$ and a meet operator $\sqcup$
- An initial value $\iota \in L$
- A transfer function associated with each statement, mapping a value of the domain to itself, $f_s : L \rightarrow L$

The data-flow graph describes the flow of data through a sequence of statements. Each node in the graph represents a statement, and directed edges between nodes represent the flow of data. We can consider the flow in the forwards or backwards direction, depending on the analysis. In the forward direction, there is an edge from statement $s'$ to statement $s$ if $s$ can immediately follow $s'$ in some execution of the program. In the backwards direction, we track how data flows from successor statements to predecessor statements, i.e., from $s$ to $s'$. The flow graph will have one distinct entry node representing $init(S_\star)$, and one or more exit nodes representing $\{final(S_\star)\}$.

For each point in the program, we represent a set of possible concrete states $\sigma$ with an abstracted state $\widehat{\sigma}$. A data-flow value $l \in L$ is then associated with each abstracted program state. Each value $l$ consists of abstracted information that we wish to extract from the program, and the set of all possible $l$ values is the domain of the analysis. On the merge of control flows, data-flow values from multiple paths are combined into one. We use the meet operator to merge the two values, summarizing the contributions of each path in one data-flow value.

The domain of values $L$ and meet operator $\sqcup$ are defined in an algebraic structure called a *semilattice*. A semilattice is a partially ordered set $L$ and binary operator $\sqcup$ with the following properties for $x, y, z \in L$:

1. $x \sqcup x = x$

2. $x \sqcup y = y \sqcup x$

3. $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$

The semilattice has a top element $\top$ with the property that $\forall x \in L, \top \sqcup x = x$, and a bottom element $\bot$ with the property that $\forall x \in L, \bot \sqcup x = \bot$. For example, if the meet operator is $\cup$, top element is $\emptyset$ since $\forall x \in L, \emptyset \cup x = x$ and the bottom element is $U$, the universal set, since $\forall x \in L, U \cup x = U$. Usually, we use $\top$ or $\bot$ to initialize data-flow values.

The transfer function $f_s$ of a statement $s$ describes how $s$ modifies data-flow values. We use $entry(s)$ to denote the data-flow value before executing the statement $s$ and $exit(s)$ to denote the value after executing $s$. In the forwards direction, we relate the data-flow values before and after executing $s$ with $exit(s) = f_s(entry(s))$.

We would like to find the values for $entry(s)$ and $exit(s)$ for each $s$ in the program. For an analysis $A$ operating in the forwards direction, we do this using the following set of equations:

$$entry(s) = \begin{cases} \iota & \text{if } s \in init(S_\star) \\ \sqcup \ \{exit(s'), \forall s' \in pred(s)\} & \text{otherwise} \end{cases}$$

$$exit(s) = f_s(entry(s))$$

The data-flow problem is to then find a solution that satisfies the set of equations on all of the $entry(s)$ and $exit(s)$. Iterative algorithms such as worklist do this by iteratively computing the information for incoming and outgoing data-flow values for each statement until there is no change in the values.

### 3.1.2 Example: Reaching Definitions

Reaching Definitions (RD) is a common data-flow analysis with a variety of applications. Our motivation for presenting it here is twofold: first, it serves as a simple example illustrating the framework described above, and second, it will aid in the understanding of one of our transformation algorithms.

The goal of RD is to determine for a program point where each variable may have been assigned, i.e., which definitions are reaching. We say a definition $d$ reaches a program point $p$ if there exists a path from $d$ to $p$ such that $d$'s value is not overwritten. For example, consider the code shown in Figure 3.1. The definition $y = 2$ reaches statements (2) and (3), but is killed by statement (4) since $y$ is reassigned to $x*y$. We would like to determine which definitions reach each node in the control-flow graph.

```
1  y = 2;
2  x = 0;
3  while(x < 5){
4        y = x*y;
5        x = x + 1;
6  }
```

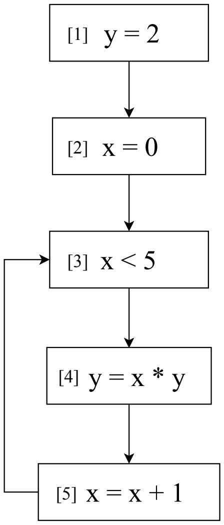


Figure 3.1: Example code with corresponding flow-graph to illustrate Reaching Definitions analysis.

The characteristics of the RD analysis are summarized in Table 3.1. The data-flow values are sets of definitions, where each definition is a tuple consisting of a program variable with a corresponding assignment statement, or ? if its assignment is unknown. For example, the data-flow value after the execution of statement (1), i.e., $exit(1)$, is $\{(x, ?), (y, 1)\}$, since $y$ was last assigned on line (1) and we do not know where $x$ was last assigned.

Table 3.1: Summary of RD data-flow problem.

| | |
|---|---|
| Domain | Sets of definitions |
| Direction | Forwards |
| Meet | $\cup$ |
| Transfer Function | $f_s = gen_s \cup (x - kill_s)$ |
| Initialize | $\{(x, ?) \mid x \in FV(S_\star)\}$ |

When multiple paths merge, i.e., when a node in the control-flow graph has multiple incoming flows/edges, we must consider the reaching definitions that are propagated from each path. Since we cannot know which path is taken during the program's execution, we must consider *all possible* reaching definitions. To include the definitions contributed from each converging path, we use set union as our meet operator. For example, the reaching definitions from (2) and (5) are propagated to statement (3). Therefore, both the definitions $(x, 2)$ and $(x, 5)$ reach the entry to statement (3).

To determine how a statement $s$ affects the data-flow values, i.e., the relationship between *exit*(s) and *entry*(s), we use the transfer function $f_s = gen_s \cup (x - kill_s)$ where $gen_s$ is the set of definitions generated by statement $s$ and $kill_s$ is the set of definitions killed by $s$. For example, the assignment statement $x = 0$ on line (2) generates the definition $(x, 2)$ and kills all other definitions of $x$, i.e., $(x, ?)$ and $(x, 5)$. For non-assignment statements, such as the while statement on line (3), $exit(s) = entry(s)$ since no definitions are generated or killed.

We initialize *exit*(s) with the data-flow value $\{(x, ?)\}\ \forall x \in FV(S_\star)\}$, i.e., the top element in the domain. In this way, we do not assume a definition $d$ reaches a statement $s$ unless we find a path propagating $d$ to $s$.

Table 3.2 shows the *kill* and *gen* sets for each statement in the example, which are computed algorithmically. For example, the statement $y = 2$ on line (1) kills all definitions of y, i.e., $\{(y, ?), (y, 1), (y, 4)\}$, and generates the definition $(y, 1)$.

Table 3.2: Computation of *kill* and *gen* functions.

| $s$ | $kill_s$ | $gen_s$ |
|---|---|---|
| 1 | $\{(y, ?), (y, 1), (y, 4)\}$ | $\{(y, 1)\}$ |
| 2 | $\{(x, ?), (x, 2), (x, 5)\}$ | $\{(x, 2)\}$ |
| 3 | $\emptyset$ | $\emptyset$ |
| 4 | $\{(y, ?), (y, 1), (y, 4)\}$ | $\{(y, 4)\}$ |
| 5 | $\{(x, ?), (x, 2), (x, 5)\}$ | $\{(x, 5)\}$ |

Once we have our *kill* and *gen* sets, we use Algorithm 1 to compute the reaching definitions for each statement in the example, or equivalently, each node in the control-flow graph. We begin by initializing the data-flow value for the *exit* of each statement to the set $\{(x, ?)\}\ \forall x \in FV(S_\star)$. We then repeat lines 5 through 10 while there is a change to $exit(s)$ for any $s \in S_\star$. In each iteration, we compute $entry(s)$ and $exit(s)$ for each $s \in S_\star$ using the data-flow equations defined in Section 3.1.1.

**Algorithm 1** Worklist algorithm to compute reaching definitions.

```
 1: function REACHING-DEF($S_\star$)
 2:     for all $s \in S_\star$ do
 3:         $exit(\mathrm{s}) = \{(x, ?)\}\ \forall x \in FV(S_\star)\}$
 4:     end for
 5:     while any change to $exit$ do
 6:         for all $s \in S_\star$ do
 7:             $entry(s) = \cup\ exit(s'), s' \in pred(s)$
 8:             $exit(\mathrm{s}) = gen_s \cup (entry(\mathrm{s}) - kill_s)$
 9:         end for
10:     end while
11: end function
```

Using Algorithm 1, we are able to compute the reaching definitions for each of the statements in the example. The solution to the data-flow problem is shown in Table 3.3.

Reaching Definitions is an example of a second order analysis, in which the abstracted program states $\widehat{\sigma}$ are sets of values. For more complex analyses, such as Parity or Sign analysis, $\widehat{\sigma}$ is a mapping from program variables to abstract values. We give an example of a first order analysis next.

Table 3.3: Solution to the RD data-flow problem.

| $s$ | *entry*(s) | *exit*(s) |
|---|---|---|
| 1 | $\{(x,?),(y,?)\}$ | $\{(x,?),(y,1)\}$ |
| 2 | $\{(x,?),(y,1)\}$ | $\{(x,2),(y,1)\}$ |
| 3 | $\{(x,2),(x,5),(y,1),(y,4)\}$ | $\{(x,2),(x,5),(y,1),(y,4)\}$ |
| 4 | $\{(x,2),(x,5),(y,1),(y,4)\}$ | $\{(x,2),(x,5),(y,4)\}$ |
| 5 | $\{(x,2),(x,5),(y,4)\}$ | $\{(x,5),(y,4)\}$ |

### 3.1.3 Example: Parity Analysis

Parity is a first order data-flow analysis that interprets an integer value as being either even or odd, or if it cannot decide, then both. If we define $\mathbf{Parity} = \{\text{even}, \text{odd}\}$ as the set of abstract values, then we say that Parity analysis operates on $V \to \mathcal{P}(\mathbf{Parity})$ abstract domain, where $V$ is the set of integer variables in a program. Commonly, $\emptyset$ and $\mathcal{P}(\mathbf{Parity})$ are denoted as $\bot$ and $\top$, respectively.

Figure 3.2 illustrates how Parity interprets a fragment of code when the incoming value of $x$ is the top value, i.e., $x \mapsto \top$. At the conditional statement, Parity determines that only even values of $x$ can enter this statement, while odd values follow the false branch. Inside the conditional statement, the analysis reasons about the effect of the statement $x = x + 3$ and determines that it changes all even values to odd. At the merge point of the two branch outcomes, the analysis combines two data-flows which results in the final $\widehat{\sigma}[x \mapsto \text{odd}]$ abstract state.

Later, in Section 4.1.1, we extend this example to demonstrate the concepts of APT which targets similar first order data-flow analyses.

```
if(x%2 == 0){
        x = x+3;
}
```

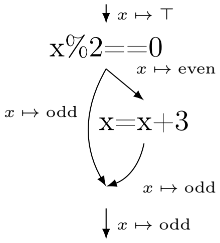


Figure 3.2: Parity interpretation of a code fragment.

## 3.2 Semantic Program Equivalence

Proving that two programs (or program fragments) are semantically equivalent is an important task that has application in various areas of computer science. For example, programmers routinely refactor their code with the goal of producing an improved version that exhibits the same behavior as the original program. Compilers also apply code transformations that aim to reduce a program's computational resources while preserving its original behavior.

### 3.2.1 Contextual Semantic Equivalence

When determining whether two program fragments are semantically equivalent, we consider contextual equivalence. That is, we say two fragments are semantically equivalent if we can substitute one for the other in any program and the observable behavior of the program is the same. For example, $2 * x$ is semantically equivalent to $x + x$ since we can substitute one for the other in any context and the behavior of the program is unchanged.

### 3.2.2 Structural Operational Semantics

To prove contextual equivalence, we must have a way to reason about a program's behavior. For this, we can use structural operational semantics, introduced by Plotkin

[32], in which a program is defined as a set of transitions between configurations of semantics. A configuration can be either a tuple containing a statement $s$ and a program state $\sigma$ (a mapping from program variables to values), or a single state $\sigma$. Thus, transitions could take either of these two forms:

$$\langle s, \sigma \rangle \to \langle s', \sigma' \rangle \text{ or } \langle s, \sigma \rangle \to \sigma'$$

The first form describes the case when execution of the first statement in $s$ results in a new configuration $\langle s', \sigma' \rangle$, where $s'$ is the rest of the program and $\sigma'$ is the updated state $\sigma$. The second form corresponds to a terminal configuration that produces state $\sigma'$ after executing $s$.

### 3.2.3 Rules for Proving Semantic Equivalence

Using the structural operational semantics, we can define statement semantic equivalence of as follows, where $\xrightarrow{*}$ indicates a multi-step reduction relation:

**Definition 1.** *Let $s_1$ and $s_2$ be two statements, then $s_1 =_{sem} s_2$ iff $\forall \sigma$, and $\langle s_1, \sigma \rangle \xrightarrow{*} \sigma'_1$ and $\langle s_2, \sigma \rangle \xrightarrow{*} \sigma'_2$, we have $\sigma'_1 \equiv \sigma'_2$.*

That is, two statements are semantically equivalent if starting at every possible program state, the execution of those two statements results in identical terminal states. We can compute those terminal states using the inductive reasoning on the structural semantics and semantics of expressions of a language.

To determine the semantic equivalence of expressions, we use the semantic functions $\mathcal{S}^\tau$ for each $\tau \in Img(\Gamma)$, where $\Gamma$ is the local type system, i.e., a typing function which determines the types of variables. The semantics functions $\mathcal{S}^\tau$ are defined by the structural operational semantics of the language and evaluate the expressions of a corresponding type to a value, i.e., $\mathcal{S}^\tau [\![ e^\tau ]\!] \, \sigma = v$. For example, an arithmetic function for integers $\mathcal{S}^{\text{Int}}$ maps an expression $e^{\text{Int}}$ to its evaluation in a state $\sigma$, if $e$ is an integer variable $x$ then $\mathcal{S}^{\text{Int}}[\![x]\!]\sigma = \sigma(x)$. Therefore, we define the semantic equivalence for expressions as following:

**Definition 2.** *Let $e_1$ and $e_2$ be two expressions of the same type $\tau$, then $e_1 =_{sem} e_2$ iff $\forall\sigma$ and $\mathcal{S}^{\tau}[\![e_1^{\tau}]\!]\ \sigma = v_1$ and $\mathcal{S}^{\tau}[\![e_2^{\tau}]\!]\ \sigma = v_2$, we have $v_1 \equiv v_2$.*

In other words, two expressions are semantically equivalent if for every possible program state, the corresponding semantics function evaluates those expressions to the same value. Therefore, for sound substitution of an expression $e_1$ with another expression $e_2$, we must show using $\mathcal{S}^{\tau}$ of the language that for every possible state $\sigma$, $e_1$ and $e_2$ evaluate to the same value. For example, given that $e_1$ is $2{*}x$ and $e_2$ is $x{+}x$, we use the semantics function $\mathcal{S}^{\text{Int}}$ which evaluates each expression to its value in the program state, that is, $\mathcal{S}^{\text{Int}}[\![2{*}x]\!]\sigma[x \mapsto X] = 2{*}X$ and $\mathcal{S}^{\text{Int}}[\![x{+}x]\!]\sigma[x \mapsto X] = X{+}X$ where $X$ represents any possible value of $x$. Since $2 * X = X + X$ for any given $X$, the two expressions are semantically equivalent and therefore, we can safely substitute one for the other in any program context.

In general, proving the semantic equivalence of two programs is a non-trivial task. Human assistance is often required to ensure program equivalence, especially after complex code refactorings. However, such strong concrete-semantic equivalence might be more than necessary for program verification techniques, such as data-flow analysis that interprets the programs at some level of abstraction. In fact, as long as the analysis computes the same information for the two program versions, we can say they are contextually equivalent in the abstraction of the analysis. In the next section, we formulate this notion of abstract-semantics equivalence.

# CHAPTER 4

# ABSTRACT-SEMANTICS PRESERVING TRANSFORMATIONS

## 4.1 Abstract-Semantics Program Equivalence

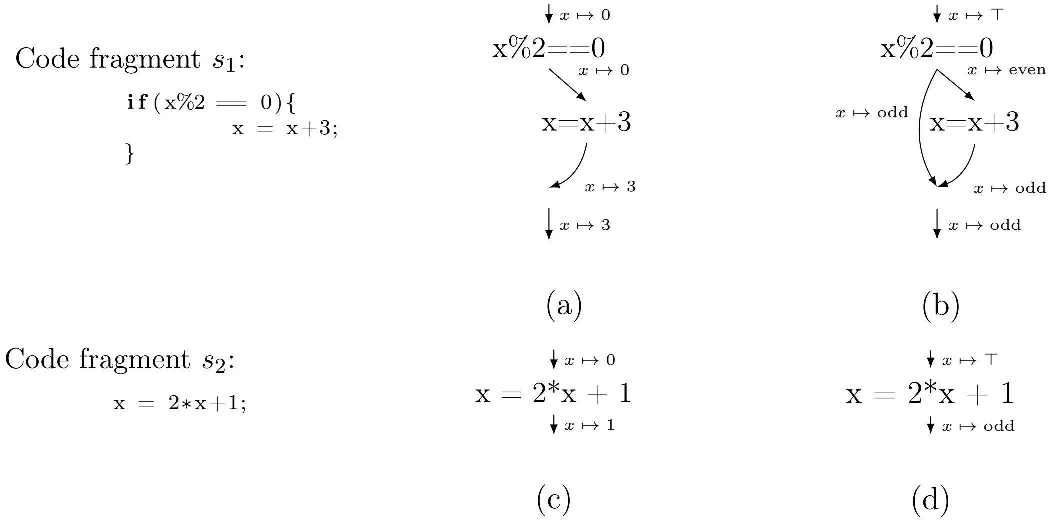
Before we present our rules for performing abstract-semantics preserving transformations, we define abstract-semantic program equivalence.

### 4.1.1 Motivating Example

Consider the two code fragments $s_1$ and $s_2$ in Figure 4.1. We can disprove the semantic equivalence of $s_1$ and $s_2$ by finding a value of `x` for which two program fragments behave differently. For example, let us examine their effect on program execution when the incoming value of `x` is zero, i.e., the incoming program state is $x \mapsto 0$. Figure 4.1(a) and Figure 4.1(c) demonstrate the execution traces of these two fragments for the incoming state $\sigma[x \mapsto 0]$. Since the resulting state from $s_1$ is $\sigma[x \mapsto 3]$ and from $s_2$ is $\sigma[x \mapsto 1]$, the two code fragments are not semantically equivalent at the concrete semantics level.

Now consider Figure 4.1(b) and Figure 4.1(d), which show how Parity interprets the same code fragments for all possible incoming values of $x$, i.e., when $\widehat{\sigma}[x \mapsto \top]$. For the conditional statement at $s_1$, Parity determines that only even values of $x$ can enter this statement, while odd values follow the false branch. Inside the conditional statement, the analysis reasons about the effect of the statement $x = x + 3$ and determines that it changes all even values to odd. At the merge point of the two branch outcomes, the analysis combines two data-flows which results in the final $\widehat{\sigma}[x \mapsto \text{odd}]$ abstract state. In the same manner Parity analyzes $s_2$, which also results in the same final state.

For the remaining elements in Parity's abstract domain, i.e., $x \mapsto \bot$, $x \mapsto$ odd or $x \mapsto$ even, the two fragments produce the same final states, which are $x \mapsto \bot$ for the first element and $x \mapsto$ odd for other elements. Since the two fragments evaluate to the same abstract state for each incoming element, we can say $s_1$ and $s_2$ are semantically equivalent in Parity abstraction.



Code fragment $s_1$:

```
if(x%2 == 0){
        x = x+3;
}
```

Code fragment $s_2$:

```
x = 2*x+1;
```

Figure 4.1: Two code fragments $s_1$ and $s_2$ which are not semantically equivalent under concrete semantics (a),(c), but are semantically equivalent under *Parity* analysis abstract semantics (b),(d).

### 4.1.2 Abstract-Semantics Equivalence

Similar to concrete-semantics equivalence, we define abstract-semantics equivalence for expressions and statements for a given analysis $A$ with the abstract domain $D_A$ and the set of transfer functions $\mathcal{F}_A$. First, we provide the definition for abstract-semantics equivalence of two statements $s_1$ and $s_2$.

**Definition 3.** *A program statement $s_1$ is semantically equivalent to a program statement $s_2$ in the abstraction of the analysis $A$, that is $s_1 =_{sem_A} s_2$, when $\forall \widehat{\sigma} \in D_A$ iff $f_{s_1}(\widehat{\sigma}) = \widehat{\sigma}'_1$ and $f_{s_2}(\widehat{\sigma}) = \widehat{\sigma}'_2$, where $f_{s1}, f_{s2} \in \mathcal{F}_A$, then $\widehat{\sigma}'_1 \equiv \widehat{\sigma}'_2$.*

That is, given any abstract state, if executing $s_1$ results in the same abstract state as executing $s_2$, the two statements are semantically equivalent in the context of the analysis.

For defining abstract-semantics equivalence for expressions, we need to introduce an additional notation $\mathcal{D}_A$ to describe the set of possible values to which $A$'s abstract semantics functions $\widehat{\mathcal{S}}^{\tau}_A$ can evaluate an expression of type $\tau$. In the case of the second-order analyses such as Reaching Definitions analysis, $\mathcal{D}_A \equiv D_A$. However, for the first-order analyses such as the *Parity* analysis, $D_A \equiv (V \to \mathcal{D}_A)$. Now we can construct the definition for abstract-semantics equivalence of two expressions $e_1$ and $e_2$.

**Definition 4.** *Two expressions $e_1$ and $e_2$ of the same type $\tau$ are semantically equivalent in the abstraction of the analysis $A$, that is $e_1 =_{sem_A} e_2$, when $\forall \widehat{\sigma} \in D_A$, iff $\widehat{\mathcal{S}}^{\tau}_A[\![e^{\tau}_1]\!]\widehat{\sigma} = d_1 \in \mathcal{D}_A$, $\widehat{\mathcal{S}}^{\tau}_A[\![e^{\tau}_2]\!]\widehat{\sigma} = d_2 \in \mathcal{D}_A$, then $d_1 \equiv d_2$.*

That is, two expressions are semantically equivalent in the abstraction of the analysis if for every possible program state, the semantics function evaluates the expressions $e_1$ and $e_2$ to the same value. For example, consider again the code fragment $s_1$ in the example above. Let $e_1$ be the original expression $x + 3$ and let $e_2$ be another expression $x + 1$. Then $\widehat{\mathcal{S}}^{\tau}_A[\![x + 3]\!]\widehat{\sigma} =$ odd, that is, the abstract semantics function for parity analysis evaluates the expression $x+3$ to odd, since $x$ is even at the current program state $\widehat{\sigma}$. Similarly, $\widehat{\mathcal{S}}^{\tau}_{A}[\![x+1]\!]\widehat{\sigma} = \text{odd}$, since adding 1 to an even value also results in an odd value. We can consider the other possible incoming abstract states $\widehat{\sigma}$, i.e., when $x$ is odd or {even, odd}. In the former case, the semantic function evaluates both $e_1$ and $e_2$ to even, and in the latter, both expressions are evaluated to {even, odd}. Thus the expressions $x + 1$ and $x + 3$ are semantically equivalent in Parity abstraction.

When the analysis cannot reason about an expression, e.g., if the variable $x$ was not resolvable in the above example, its evaluation is over-approximated with the top element. Also note that not all statements and expressions modify the abstract state, i.e., the data-flow information. For example, adding a print statement to the example code snippet would not affect its behavior according to the analysis, and hence does not modify any abstract state. Next, we use these observations to define rules which allow us to perform sound program transformations.

## 4.2 Abstract-Semantics Preserving Transformations

We now present rules for performing APT on a set of open-source Java programs to produce a set of real-world benchmarks for suitable for SPF. We then use these rules to construct two transformation algorithms, the first of which removes external dependencies from a target program and the second eliminates nonlinear symbolic expressions in the program.

### 4.2.1 Removing External Dependencies

We assume there is a Java class file $P$, i.e., a program, which is compilable in a type system $\Gamma_1$ with the set of types $T_1$ of the project it resides in. $P$ might have several Java classes defined in it, i.e., it defines $T_P$ types. In addition, SPF has a local type system $\Gamma_2$ with the set of types $T_2$ in which we would like $P$ to be compilable, i.e., the types defined in SPF or the Java standard library. Thus, $T_1 \cap T_2 \supseteq T_P$, that is, the local type system can resolve the types defined in $P$.

### Rules

We now present rules for removing statements, i.e., replacing with `skip`, and substituting expressions in $P$ using the notion of abstract-semantics equivalence. The two rules are shown in Figure 4.2.

$$\begin{array}{lll} \textit{stmt} & s_1 \longrightarrow \text{skip;} \\ & \quad \text{if } \forall \widehat{\sigma} \in \mathcal{D}_{SE} \text{ and } f_{s_1} \in \mathcal{F}_{SE},\ f_{s_1}(\widehat{\sigma}) = \widehat{\sigma} \text{ and } \exists x \in Var(s_1) \text{ s.t. } x \notin \text{Dom}(\Gamma_2) \\ \textit{expr} & e_1^{\tau} \longrightarrow e_2^{\tau} \\ & \quad \text{if } \widehat{\mathcal{S}}_{SE}^{\tau}[\![e_1^{\tau}]\!] = d \in D_{SE} \text{ and } \tau \in T_2 \text{ and } \exists x \in FV(e_1^{\tau}) \text{ s.t. } x \notin \text{Dom}(\Gamma_2) \\ & \quad \text{and } \widehat{\mathcal{S}}_{SE}^{\tau}[\![e_2^{\tau}]\!] = d \text{ and } \forall x \in FV(e_2^{\tau}), x \in \text{Dom}(\Gamma_2) \end{array}$$

Figure 4.2: Transformations rules for statements (stmt) and expressions (expr) to resolve unknown types.

The first rule *stmt* says we can remove a statement, i.e., replace it with `skip;`, if the transformer for $s_1$ does not change the abstract state and the variables either defined or used in the statement cannot be resolved by SPF, i.e., the local type system. Through a series of substitutions and post-processing, our algorithm eliminates dependencies on variables defined in the removed statements. However, the *stmt* rule keeps statements that preserve the abstract state but either define or use variables of types present in $\Gamma_2$, such as `System.out.println()`. We do so to maintain as much of the original program structure as possible for easy reference to the original code and to reduce the complexity of our transformation algorithm.

The second rule *expr* says that an expression $e_1$ of type $\tau \in T_2$ can be substituted with another expression $e_2$, if $e_1$ has at least one free variable with a type that $\Gamma_2$ cannot resolve, and the abstract expression semantics for $\tau$ always evaluates $e_1$ and $e_2$ to the same abstract value. Even though $\Gamma_2$ cannot resolve types of some free variables, the abstract expression semantics should be able to reason about the expression based on the expression category, i.e., a method invocation or field access. In general, if an analysis cannot reason about an expression category, then its evaluation is over-approximated with an element from its abstract domain, for example with $\top$. Since there is at least one free variable in $e_1$ for which the local type system cannot determine the type, then we cannot use $\Gamma_2$ to determine the type of $e_1$. Our algorithm infers the type of $e_1$ based on the program context.

For example, consider the listing below. If $\Gamma_2$ cannot resolve the type of `contextStack`, we can infer its boolean type based on its use in the conditional statement.

```
if(contextStack.isEmpty()){
    ...
}
```

```
if(newSymbolicBoolean()){
    ...
}
```

Figure 4.3: Substitution of an expression which has inferred boolean type.

In another example, we reason about the type of `n.value()` based on its comparison to a floating-point literal.

```
while(n.value() < 10.5){
    ...
}
```

```
while(newSymbolicDouble() < 10.5){
    ...
}
```

Figure 4.4: Substitution of an expression which has inferred double type.

We have predefined expressions $e_2$ for each type $\tau$ such that it evaluates to the same abstract value and resolvable by SPF. These values are shown in Table 4.1. For example, in the first listing, we would substitute an unresolved expression of type `Boolean` with the expression `Debug.makeSymbolicBoolean()` where `Debug` is a type defined in $\Gamma_2$.

Table 4.1: Predefined expressions used for substitution.

| $\tau$ | $e_2^\tau$ |
|---|---|
| boolean | Debug.makeSymbolicBoolean() |
| byte | Debug.makeSymbolicByte() |
| char | Debug.makeSymbolicChar() |
| double | Debug.makeSymbolicReal() |
| float | Debug.makeSymbolicReal() |
| integer | Debug.makeSymbolicInteger() |
| long | Debug.makeSymbolicLong() |
| short | Debug.makeSymbolicShort() |
| string | Debug.makeSymbolicString() |

### Transformation Algorithm

Algorithm 2 presents the pseudo code for removing $P$'s dependency on $\Gamma_1$ while preserving $P$'s behavior in the abstraction of symbolic execution. The algorithm takes as an input the program or Java class file $P$.

From line 2 through line 32, the algorithm iterates over each of the classes defined in $P$ and each of the classes' methods (lines 3 through 25). In the current implementation, methods that have unresolvable parameter types are removed (lines 4 through

**Algorithm 2** Dependency removal algorithm for a java file $P$ to be compilable in $\Gamma_2$ while preserving operations on essential types $T_{SE}$.

```
function TRANSFORM($P$)
    for all $cl \in$ classes($P$) do
        for all $m \in$ methods($cl$) do
            if hasUnresolvableParamTypes($m$) then
                removeMethod($cl$, $m$)
                continue;
            end if
            for all $s \in$ statements($m$) do
                $\mathit{stmt}(s)$
            end for
            for all $e \in$ reversePre-Order(expressions(m)) do
                $\tau =$ inferType($e$)
                if $\tau \in T_{SE}$ then
                    $\mathit{expr}(e^{\tau})$
                end if
            end for
            for all $x \in$ fieldVars($m$) do
                $\tau =$ getType($x$)
                if $\tau \in T_{SE}$ then
                    $s_{\text{def}} =$ defineAndInitialize($x$);
                    insert($m$, $s_{\text{def}}$)
                end if
            end for
            updateReturnType($m$)
        end for
        removeFields($cl$)
        for all $\tau_{\text{super}} \in$ superTypes($cl$) do
            if $\tau_{\text{super}} \notin T_2$ then
                removeSuperType($cl$, $\tau_{\text{super}}$)
            end if
        end for
    end for
    for all $\tau_{\text{import}} \in$ import($P$) do
        if $\tau_{\text{import}} \notin T_2$ then
            remove(s)
        end if
    end for
end function
```

7), as well as invocations of the removed methods. In the future, we planned to keep all of the methods, removing parameters of unresolvable type and updating the corresponding method invocations to match.

For methods with resolvable parameter types, the algorithm iterates over its statements in lines 8 through 10. For each statement, it first checks whether the rule *stmt* could be applied, that is, whether the statement can be removed. On the updated set of statements, the algorithm arranges the expressions in the post-order traversal of the expression node of the method's AST. We enforce such order so the leaf expressions with types unresolved by $\Gamma_2$ are substituted first, which maintains more of the overall structure. For example, assume in the following code that $\Gamma_2$ can resolve `x` and `y` but not `obj`. Rather than replacing the whole conditional statement with `newSymbolicBoolean()`, we substitute `obj.size()` with `newSymbolicInteger()`.

```
if(obj.size() < 5 && x < y){
    ...
}
```

```
if(newSymbolicInteger() < 5 && x < y){
    ...
}
```

Figure 4.5: Effect of substituting expressions during the post-order traversal of the method's AST.

Before checking whether the rule *expr* could be applied, the algorithm infers the type of the expression based on its usage context on line 12. For example, if $e$ is used on the right-hand side of an assignment statement that defined a variable with an integer type, then the algorithm infers that $e$'s type is `Int`.

After iterating over the expressions and performing necessary transformations (line 13 through 15), the algorithm finds class variables used in the method. For each class variable, it generates a predefined definition and initialization statement and inserts them into $m$'s body. In this way, the class variables become local variables. We do this to contain all of $m$'s relevant code in the method body. This way, our framework could be extended to extract only relevant methods out of open-source projects, i.e., producing individual benchmark methods rather than benchmark classes. (Recall we focus on intra-procedural analysis.)

On line 24, the algorithm ensures that the declared return types of the methods match. For example, the method might return a type that is not in $T_2$, thus line 9 removes such statement. In this case, line 24 changes the return type of $m$ to `void`. This completes modification steps for methods.

On lines 26 through 31, the algorithm completes modification of all classes in $P$. All of the field variables are removed, and then unresolved super class types are removed. Finally, the algorithm finishes by removing from $P$ unresolved import statements.

### 4.2.2 Substituting Nonlinear Symbolic Expressions

While resolved dependencies make a method compilable, it still might not be suitable for SPF. For example, nonlinear expressions in PCs could cause SPF to fail or be inefficient. In general, the decision problem for arbitrary path constraints is undecidable. However, solving constraints in the theory of linear integer arithmetic is decidable, while solving constraints involving nonlinear integer arithmetic is undecidable.

In practice, a solver supporting nonlinear integer arithmetic tries to process such a constraint and returns "unknown" if it cannot decide it in the alloted time. In this way, the constraint is over-approximated with the top element since it cannot be evaluated by the analysis. However, the symbolic execution engine has to wait for the solver to process the constraint, returning "unknown" only after it has timed out (or throwing an exception if the solver does not support that class of arithmetic). Alternatively, we can anticipate expressions that the solver may not be able to process, and replace them with an equivalent over-approximation, thus improving the efficiency.

To do so, we define a rule and corresponding algorithm for replacing nonlinear expressions with over-approximated expressions, in particular, new symbolic variables. A simple example of this is shown in Figure 4.6. The nonlinear symbolic expression `a*b` would generate a nonlinear path constraint, so we introduce a new symbolic variable `sym` and use this variable to over-approximate the expression.

```
void compute(int a, int b){
    int c = a*b;
    ...
}
```

```
void compute(int a, int b) {
    int sym = newSymbolicInteger();
    int c = sym;
    ...
}
```

Figure 4.6: A nonlinear expression is replaced with a symbolic variable.

### Rule

Figure 4.7 shows our rule *ns_expr* for substituting a nonlinear symbolic expression with a symbolic variable. Note the similarities between this rule and the *expr* rule introduced in the previous section. Both allow us to perform a substitution of an expression $e$ as long as the expression evaluates to the same abstract value. The difference is that in the *ns_expr* rule, we substitute an expression $e$ of type $\tau$ with a symbolic variable of the same type when the expression $e$ is a nonlinear symbolic expression. We define a nonlinear symbolic expression as one whose left-hand operand and right-hand operand both contain variables, and the expression operator is multiplication, division or the remainder operator.

$$\begin{array}{llll} ns\_\,expr & e_1^\tau & \longrightarrow & e_2^\tau \end{array}$$

$$\text{if } \widehat{\mathcal{S}}^{\tau}_{SE}[\![e_1^\tau]\!] = d \in D_{SE} \text{ and } \tau \in T_2 \text{ and } FV(lhs) \neq \emptyset \text{ and } FV(rhs) \neq \emptyset$$

$$\text{and } op \in \{*, /, \%\} \text{ and } \widehat{\mathcal{S}}^{\tau}_{SE}[\![e_2^\tau]\!] = d$$

Figure 4.7: Transformation rule for nonlinear symbolic expressions.

We correspond each nonlinear symbolic expression with a new symbolic variable that we introduce into the program, i.e., an expression $e_1^\tau$ of the form $(e_i\ op\ e_j)$ is replaced with an expression $e_2^\tau$ of the form $sym_{(e_i\ op\ e_j)}$.

### Transformation Algorithm

Algorithm 3 presents the pseudo code for removing $P$'s nonlinear symbolic expressions. From lines 2 through 29, the algorithm iterates over each of the classes defined in $P$. For each method in the class, the algorithm first identifies all of the nonlinear symbolic expressions and replaces them with symbolic variables (line 5 through 7). As before, this is done in the post-order traversal of method's AST. This maintains the relationships between any such expressions in the method body. For example, given that `a`, `b`, `c` and `d` are symbolic, `(a*b)/(c*d)` is replaced with `sym1/sym2` which is then repalced with `sym3`. Then if `a*b` appears later in the program it can be replaced with `sym1`.

Next, we reassign symbolic variables after statements that kill their corresponding expressions (line 13 through 19). For example, consider the following code. The assignment statement `b = b + 1` "kills" any expression using the variable `b` since its value has changed, e.g., `a*b`. To preserve the intentions of the original code, we reassign the symbolic variable substituted for `a*b` after the expression is killed. Thus, we create a new symbolic variable after the line `b = b + 1`, which SPF interprets as a new symbolic state.

```
void compute(int a, int b){
    int c = a*b;
    ...
    b = b + 1;
    ...
    c = a*b;
}
```

```
void compute(int a, int b) {
    int sym = newSymbolicInteger();
    int c = sym;
    ...
    b = b + 1;
    sym = newSymbolicInteger();
    ...
    c = sym;
}
```

Figure 4.8: A symbolic variable is reassigned after its corresponding expression is killed.

Once all of the new assignment statements have been inserted, we use Reaching Definitions to remove assignments that are never used. We implemented this analysis using a constraint based approach, incorporating the control flow of the program into the constraints with the equations defined below. We then solved the constraints using the iterative work-list algorithm.

$$entry(s) \subseteq \begin{cases} \iota & \text{if } s \in init(S_\star) \\ \cup \ \{exit(s'), \forall s' \in pred(s)\} & \text{otherwise} \end{cases}$$

$$exit(s) \subseteq f_s(entry(s))$$

On lines 20 through 24, we collect the reaching definitions for the statements that originally contained nonlinear symbolic expressions, i.e., statements that now have new symbolic variables. Any assignment statement to a symbolic variable that is not in this set of reaching definitions can then be removed (line 27). In this scenario, the expression that the symbolic variable represents is killed before the symbolic variable is used.

The following code listing demonstrates this step. We introduce a new symbolic variable `sym` to replace the nonlinear symbolic expression `a*b`. Inside the conditional statement, the variable `b` is reassigned, killing the expression `a*b`. Thus we insert an assignment of `sym` immediately after the expression is killed. However, before the value of the symbolic variable is used, the expression is killed again with the assignment statement `a = 2*a`. By the time we reach the last line `sym`, only the assignment to `sym` immediately after `a = 2*a` is reaching. Thus we can remove the assignment of `sym` after the statement `b = b + 1`.

```
void compute(int a, int b){
    int c = a*b;
    if (c < b) {
        b = b+1;
    }
    a = 2*a;
    c = a*b;
}
```

```
void compute(int a, int b){
    int sym = newSymbolicInteger();
    int c = sym;
    if (c < b) {
        b = b+1;
        ~~sym = newSymbolicInteger();~~
    }
    a = 2*a;
    sym = newSymbolicInteger();
    c = sym;
}
```

Figure 4.9: Removal of a non-reaching definition.

Note that if the assignment statement `a = 2*a` was in an `else` block, both definitions would be reaching. Since there exists a path from both definitions to the statement `c = sym`, neither would be removed, as shown in the code listed below.

```
void compute(int a, int b){
    int c = a*b;
    if (c < b) {
        b = b+1;
    } else {
        a = 2*a;
    }
    c = a*b;
}
```

```
void compute(int a, int b){
    int sym = newSymbolicInteger();
    int c = sym;
    if (c < b) {
        b = b+1;
        sym = newSymbolicInteger();
    } else {
        a = 2*a;
        sym = newSymbolicInteger();
    }
    c = sym;
}
```

Figure 4.10: Example illustrating possible reaching definitions.

**Algorithm 3** Nonlinear expression removal algorithm for SPF to more effectively analyze a java file $P$.

```
function REMOVENONLINEARSYMBEXPR($P$)
    for all $cl \in$ classes($P$) do
        for all $m \in$ methods($cl$) do
            for all $e \in$ reversePre-Order(infixExpressions(m)) do
                if containsVar(lhs) && containsVar(rhs) && op $\in \{*, /, \%\}$ then
                    replaceWithSymbVar($e$)
                end if
            end for
            for all $x \in$ newSymbVar($m$) do
                $s_{\mathrm{def}}$ = defineAndInitialize($x$);
                insert($m$, $s_{\mathrm{def}}$)
            end for
            for all $s \in$ statements($m$) do
                $ks_s$ = computeKillSet(s)
                for all $e \in ks_s$ do
                    $sym_{assign}$ = reassignSymVarForExpr(e)
                    insertAfter($sym_{assign}$, s)
                end for
            end for
            for all $s \in$ statements($m$) do
                if containedNonlinearSymbExpr($s$) then
                    $RD = RD \cup$ computeReadingDef(s)
                end if
            end for
            for all $assign_{sym} \notin RD$ do
                remove($assign_{sym}$)
            end for
        end for
    end for
end function
```

# CHAPTER 5

# IMPLEMENTATION OF THE APT TOOL

In this chapter, we present an overview of our framework, then give implementation details of each of the framework's main components.

## 5.1 Overview of the Framework

Figure 5.1 shows a diagram of our implementation. The input is a CSV file of GitHub repositories as gathered by RepoReaper, and the output is a directory of benchmark programs. The framework is divided into three stages. The first stage filters the projects for files that are suitable for the analysis. This is done with two components, a Project Filter and a File Filter. The second stage transforms the resulting programs with three components, Compile, Transform, and Recompile. Last is the verification stage, in which we run SPF with the generated class files to ensure that they can be analyzed. Below we describe each component in detail.

## 5.2 Extracting Programs

### 5.2.1 Project Filter and Downloader

We begin with a dataset of GitHub repositories obtained from RepoReaper [26], a software mining tool that computes several meta-data for GitHub projects to determine whether a repository contains an engineered software project or a toy

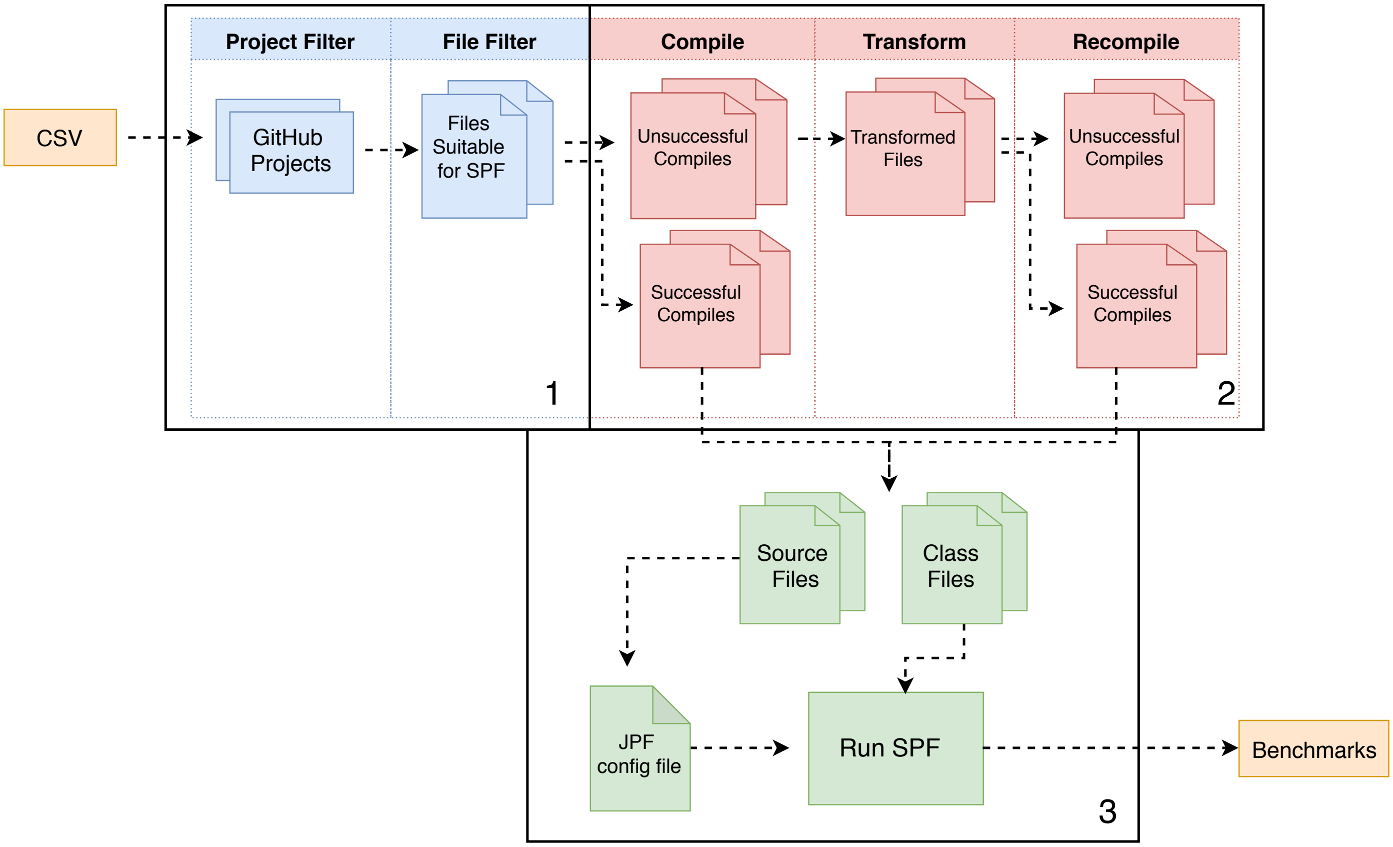


Figure 5.1: Workflow of the APT tool.

project such as a homework assignment. We use this dataset because it is large and includes meta-data about each repository, including the programming language that the project uses. With this dataset, we identify projects written in Java that have a minimum and maximum number of lines of code specified by the user. We then download a user-specified number of projects that meet the filtering criteria using the Git URLs provided by the RepoReaper dataset. Projects that are already in our database are not downloaded again.

This component is implemented in Java with approximated 750 lines of code and 6 classes.

### 5.2.2 File Filter

We search the downloaded projects for Java files that have characteristics meaningful for evaluation with SPF. Specifically, we search for files that contain methods suitable for SPF. We define suitable methods as those which have integer-only parameters and at least one integer operation in the method's body. If a class file contains at least one such method, it is chosen as a potential benchmark program and copied to a separate location for further processing.

This static analysis is implemented on the AST level using Eclipse JDT's API for creating and manipulating ASTs. We do both our static analysis and code manipulation on this level since the AST excludes syntax details irrelevant to our analysis while maintaining program structure and content. Our file filter iterates over each Java file in the in obtained projects and matches its AST elements with our specified criteria. The AST provides for each method a list of parameters with their types, so we can easily check for methods with integer parameters. To filter for methods with integer operations, we created a Visitor which traverses each method's AST and identifies integer operations. The filtered source codes become benchmark candidates.

The File Filter component is implemented in Java with approximated 1K lines of code and 5 classes.

## 5.3 Applying Transformations

### 5.3.1 Compile

To determine whether the potential benchmark programs require further transformation processing, we compile those files as stand-alone benchmark programs. This is done from the application using the `exec` system call. The files that successfully compile are set aside as benchmark-ready programs; those that do not compile are passed along to the Transform component.

### 5.3.2 Transform

For each file passed to the Transform component, Algorithm 2 is first applied to remove its external dependencies while preserving code structure relevant to the analysis. Algorithm 3 is then applied to remove any nonlinear symbolic expressions. Both algorithms operate on the AST level, implemented with the Eclipse JDT's API. We use the Visitor design pattern to traverse the tree, which allows us to iterate over each program's classes, methods, statements and expressions in the required order and apply the necessary deletions/substitutions.

The Transform component (excluding Reaching Definitions) is implemented in Java with approximately 3K lines of code and 15 classes. Reaching Definitions alone is implemented with 2K lines of code and 14 classes.

#### Testing

To evaluate the correctness of Algorithm 2, we created 11 test cases for each instance of program transformations such as: removal of a static method invocation, removal of a variable declaration, and substitution of a method invocation (whose return value is an integer) with a symbolic integer. To determine the correctness of the program transformations we manually compared the results of SPF on the original and the transformed programs and verified that they are equivalent.

To evaluate the correctness of Algorithm 3, we created two test suites, one to test our substitution of nonlinear symbolic expressions, and another to test our implementation of Reaching Definitions analysis. The first test suite contains methods with nonlinear symbolic expressions. The expressions appear in a variety of Java statements, e.g., method invocations, assignment statements, conditional statements, etc. Some of the nonlinear expressions are operands of larger expressions, and some are repeated. We ran lines 4 through 8 of Algorithm 3 on this suite, and manually verified that the correct substitutions were made. The second test suite contains methods with different control flows constructs (e.g., loops, conditional statements, nested blocks, etc.). We manually computed the reaching definitions at various program points, and verified that they were equivalent to the reaching definitions that our analysis computed.

### 5.3.3 Recompile

Once the transformation is complete, the files are compiled again. At this point, we record the total number of files that we could successfully compile, including those which did not require transformation. Some files still may not compile after transformation and this could be for two reasons: they contain features that are not handled in our implementation, or they weren't compilable in the original project. For these files, we log their compilation error(s) for further consideration.

## 5.4 Verification and Preparing for SPF

In the Transform component, we verified that the resulting source code files can be compiled after applying the necessary transformations. Additionally, we run the resulting benchmarks with SPF to ensure that they can be processed. To do so, we have to prepare them for SPF analysis. Since SPF requires the main method to serve as the entry point for a method analysis, our framework instruments each benchmark program with a main method, as well as a call from main to each of the program's SPF suitable methods. An example of an original class and its instrumented version are shown in Listings 5.1 and 5.2, respectively. This instrumentation is done on the bytecode level using the Apache Commons Byte Code Engineering Library (BCEL).

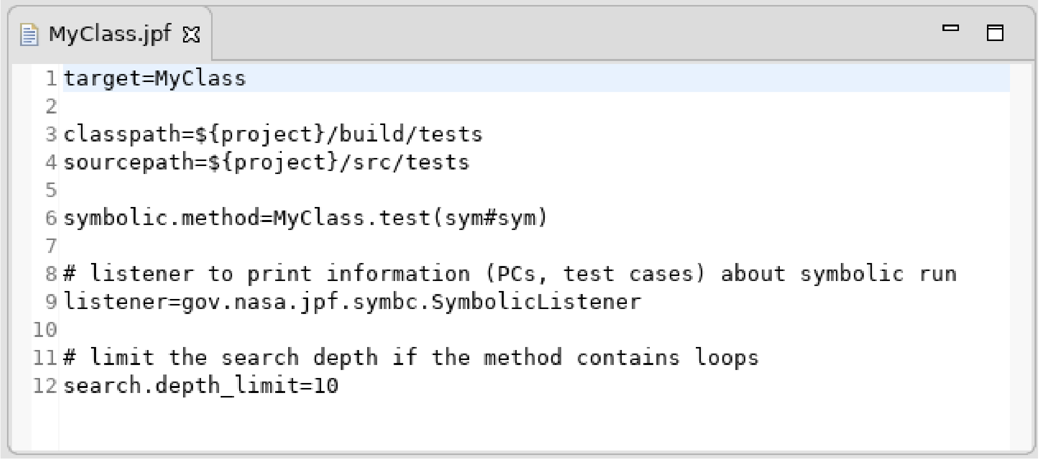
We then auto-generate program-specific Java PathFinder configuration files, specifying the method(s) to run the analysis on and the variable(s) to execute symbolically. For methods that contain loops, we also assign a bound on the number of path conditions checked, since the number of branch constraints generated grows exponentially in the presence of loops. The configuration file for the example code is shown in Figure 5.2.

Listing 5.1: Original Class.

```
public class MyClass {
  public void test(int a, int b) {
    int c = a - b;
    while (c > 0){
        ...
    }
  }
}
```

Listing 5.2: Instrumented Version.

```
public class MyClass {
  public void test(int a, int b) {
    int c = a - b;
    while (c > 0){
        ...
    }
  }
  public static void main(String[] args) {
        test(0, 0);
  }
}
```



```
target=MyClass

classpath=${project}/build/tests
sourcepath=${project}/src/tests

symbolic.method=MyClass.test(sym#sym)

# listener to print information (PCs, test cases) about symbolic run
listener=gov.nasa.jpf.symbc.SymbolicListener

# limit the search depth if the method contains loops
search.depth_limit=10
```

Figure 5.2: Java PathFinder configuration file for the example code.

# CHAPTER 6

# EVALUATION

To evaluate the effectiveness and impact of APT, we use our tool to obtain SPF-suitable benchmarks from GitHub repositories, computing the percentage by which we were able to increase the number of compilable benchmarks in those repositories. We then use this dataset to investigate how a larger set of benchmarks affects the replication of a study previously conducted with a handful of programs. In particular, we reproduce an evaluation of the Green solver framework and compare our conclusions to those of the original study. In the following sections, we present our generated dataset and the results of the Green solver case study, which we use to answer the following research questions:

**RQ1 (effectiveness).** Can we use open-source repositories to generate a suite of compilable benchmark programs tailored specifically to SPF?

**RQ2 (impact).** Can a larger set of benchmark programs bring new insight to existing evaluations of symbolic execution techniques?

## 6.1 Generating Benchmarks for Symbolic PathFinder

We first give an overview of the dataset we generated using our tool. We perform our experiments on an Intel 3.50GHz Xeon E5-1620 workstation running CentOS version 7 and Java version 8.

### 6.1.1 Input

We use RepoReaper to identify and download 1000 Java projects that have a minimum and maximum number of lines of code of 100 and 10,000, respectively. The RepoReaper dataset was retrieved January 7, 2019, and the projects were downloaded July 25, 2019.

### 6.1.2 Generated Dataset

The results of running our tool with the supplied input are show in Table 6.1. The first column specifies the number of GitHub projects that were downloaded from the RepoReaper dataset. Columns 2-4 describe the collective files found in those projects and sub columns labeled “C” and “M” show the number of classes and the number of methods, respectively. The column labeled “Compilable After Transform” shows data for the final set of benchmark programs and the last column shows data for the programs which were still not compilable. For example, for 50 GitHub projects that contain 1.3K classes and 8K methods, we determined that 82 classes containing 144 methods are potential benchmark candidates. From that set, 17 classes containing 41 benchmark methods could be compiled on their own. The rest of 103 methods in 65 classes require further processing. After applying our transformation component and verifying the resulting programs, the total number of benchmark methods became 88 in 55 classes. The rest of 56 methods in 27 classes our implementation could not yet resolve the dependencies. This is because we are not yet handling all Java language features. For example, switch statements, try/catch blocks, throw statements and arrays are not yet fully supported.

To ensure the experimental data that we obtain is representative, we performed experiments on five datasets of different sizes. We select GitHub projects in the order which they appear in the dataset generated by RepoReaper, thus the data in rows 2-4 subsume that data in the previous row.

Table 6.1: Results of transforming open-source programs into SPF benchmarks.

| GitHub Projects | Total | | Suitable For SPF | | Compilable As Is | | Compilable After Transform | | Unsuccessful Compiles | |
|---|---|---|---|---|---|---|---|---|---|---|
| | C | M | C | M | C | M | C | M | C | M |
| 50 | 1.3K | 8K | 82 | 144 | 17 | 41 | 55 | 88 | 27 | 56 |
| 100 | 2.3K | 15K | 115 | 211 | 21 | 48 | 77 | 122 | 38 | 89 |
| 250 | 5K | 29K | 268 | 486 | 47 | 96 | 143 | 244 | 123 | 242 |
| 500 | 10K | 64K | 566 | 1034 | 134 | 254 | 330 | 504 | 236 | 530 |
| 1000 | 20K | 125K | 1091 | 2032 | 237 | 448 | 611 | 902 | 480 | 1130 |

### 6.1.3 Discussion

For each set of GitHub projects, we were able to increase the number of compilable, SPF-suitable classes and as a result, the number of methods that can be used as benchmarks. Starting with 50 GitHub projects, we increased the number of SPF-suitable methods by 124.6%. This rate of increase is consistent as the number of project repositories increases, with an average increase of 130.4% and a standard deviation of 24.8%.

If we did not do any transformations, we would need to filter from significantly more GitHub repositories to produce the same number of benchmark methods that we would get after transformation. For example, we would need around 500 GitHub repositories to produce 254 benchmark methods without transformation, while our approach can produce the same number of methods from half the number of repositories.

Although we focus on intra-procedural analyses, we also record the increase of compilable SPF-suitable classes. This is because a researcher may prefer benchmark methods that are spread over many classes, suggesting a wider variety of methods found in real-world applications. The average rate of increase of benchmark classes is 199.7% with a standard deviation of 44.0%. The average number of suitable methods per class is 1.6, thus our approach provides a variety of benchmark methods representative of those that would be found in real programs.

**RQ1.** The answer to our first research question is positive: we can generate a suite of compilable benchmark programs suitable for SPF from open-source repositories.

## 6.2 Case Study: Using Generated Benchmarks to Evaluate Green

This section presents a case study to demonstrate the potential impact a larger set of programs has on SPF evaluations. In the study, we replicate an evaluation of the Green [43] solver framework, which was originally performed on only a handful of programs. We then compare the results of our evaluation to those of the original study. We conclude that a benchmark set with an increased number of programs does in fact provide additional insights to the evaluation and hence the derived conclusions.

### 6.2.1 Overview of Previous Evaluation

The authors of Green framework noticed that applications such as symbolic execution issue to SMT solvers many queries with conceptually identical constraints. Thus if constraints can be partitioned into equivalent classes, the number of those classes could be relatively small. The satisfiability results of each equivalent class can then be stored in a database for quick retrieval.

The main challenge of this approach is to define an equivalence class and determine whether a given constraint belongs to an existing equivalence class. Green overcomes this challenges by extracting only relevant predicates in a constraint using slicing, and translating constraints to their standard form using canonization. Constraint slicing carries two benefits: it reduces the size of the constraint, but more importantly for Green, it increases the chances of finding the constraint in the cache. Canonization involves reordering constraint variables and expressing each constraint predicate in a normal form, i.e., $a_1x_1 + a_2x_2 + \cdots + a_nx_n + b \bowtie 0$, where $x_i$ are integer variables, $a_i$, $b$ are integer coefficients, and $\bowtie \in \{=, \neq, \leq\}$.

Green uses the Redis database to store solved constraints and the Jedis interface to interact with it. Since Redis is optimized to handle strings as key-value pairs, Green uses string representations for both constraints and their satisfiability results to store as the key and the value, respectively, in the database. Green supports two underlying constraint solvers: Choco [16] and CVC3 [3].

We employ two of the four performance metrics used by the authors of Green (to simplify our evaluation, we do not consider model counting): 1. a time ratio $T_s = \frac{t_+}{t_-}$ where $t_-$ is the running time of SPF when Green is not used, and $t_+$ is the running time when the Green is used, and 2. a reuse ratio $R_s = \frac{n_- - n_+}{n_-}$ where $n_-$ is the number of invocations of the decision procedure when Green is not used, and $n_+$ is the number of invocations when Green solver is used.

The time ratio measures how much (if at all) Green speeds up the analysis (e.g., a time ratio of 0.5 means the analysis was twice as fast when Green was used). The reuse ratio gives the fraction of SAT queries that do not need to be re-calculated. For example, a reuse ratio of 1.0 means all of the constraints were already available in the cached database.

The original evaluation investigates the effectiveness of Green on satisfiability queries when used (1) across program runs, (2) across different programs, and (3) across different tools that utilize SMT solvers. Our evaluation is most closely related to the authors' approach of evaluating Green across different programs, so we replicate the *across different programs* study. The results of the original evaluation indicate that programs with common functionalities share a significant amount of identical queries. The results also show that even unrelated programs share many constraints, and in particular, when analyzing a sequence of unrelated programs, the $R_S$ *tends to increase for the next program in the sequence.* Therefore, with the increase of $R_S$, the corresponding $T_S$ should decreases. However, the original study uses only six programs to arrive at this conclusion.

In our replicated study, we compute the time ratio $T_S$ and the reuse ratio $R_S$ for methods obtained from open-source repositories. We use these performance metrics to derive conclusions about (a) connections between program position in the sequence and its reuse ratio, and (b) the relationship between $R_S$ and $T_S$. We then compare our conclusions to those of the original study.

### 6.2.2 Experimental Setup

Using the benchmarks produced by our tool, we retain those methods in which SPF makes at least two calls to a constraint solver during analysis. This resulted in reducing the set of benchmarks from 902 methods in 611 classes to 151 methods in 94 classes. Although we only require two calls to a solver, many of these methods are more constraint-heavy. For example, thirty require six or more calls to a constraint solver. As stated earlier, our experiment is most closely related to the authors' approach of evaluating Green across different programs. However, while they know ahead of time whether or not their programs share similar functionality, we cannot guarantee our benchmarks share any functionality, nor can we guarantee they are unique.

To compare the performance of traditional SPF to SPF using the Green solver on analyzing the program benchmarks generated by our tool, we run each of the methods in our benchmark suite twice with SPF. The first time we use SPF's default configuration with CVC3 as the solver. The second time we instruct SPF to use the Green solver framework with a registered decision procedure, constraint solver, canonizer, and slicer. This is done by setting the option `symbolic.green = true` and specifying the services to be performed as `(slice (canonize Choco CVC3))` in the Java PathFinder configuration file. At the time of our experiment, the slicer component appears not to be working. [1]

To compute $T_s$, we average $t_-$ and $t_+$ over three runs before calculating the ratio. We obtain $n_-$ and $n_+$ values from the Green listener report. We track each method by its project, package and class of origin.

Since the order in which methods are analyzed could affect the constraints available for reuse, and hence $T_s$ and $R_s$ values, we initially considered four different method orderings: $S1$, $S2$, $S3$ and $S4$. In the sequence $S1$, methods are grouped together by class, classes are grouped by packages, and packages are grouped by projects. Projects, packages and classes are listed in lexicographical order. The sequences $S2$, $S3$ and $S4$ are three random orderings of the methods in $S1$.

For each sequence, we ran methods with SPF one after another and collected the necessary information to produce the statistics for $T_s$ and $R_s$. We then calculated the

[1] When trying to reproduce an example in the paper as well as in our experiments and noticed the slicer has no effect. We notified Green developers.

t-test for the means between each pair of sequences' $T_s$ scores and similarly each pair of sequences' $R_s$ scores. The t-test shows if there is a significant difference between the means of two datasets. The results indicate no significant difference between the sequences' $T_s$ and $R_s$ variables. For example, the t-test between $S1$ and $S2$ for $T_s$ results in a p value of 0.76, and for $R_s$ a p value of 0.82. The t-test between $S2$ and $S3$ for $T_s$ yields a p value of 0.84, and for $R_s$ a p value of 0.62. Since all four sequences resulted in similar distributions of $R_s$ and $T_s$, we show the results of only one random sequence.

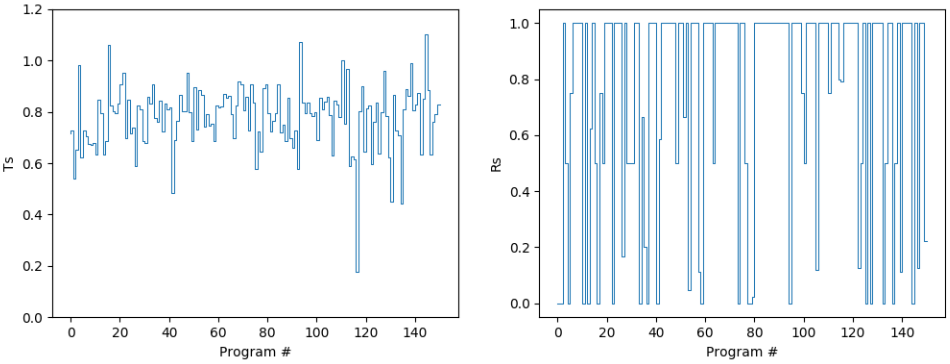
### 6.2.3 Results

We first discuss the results of the entire dataset (i.e., plots of $T_s$ and $R_s$ for every method in the sequence). Since we failed to observe the same trends as in the original study, we then focus on constraint-heavy methods, which we define to have six or more PCs, and discuss our findings for this subset of programs.

#### All methods

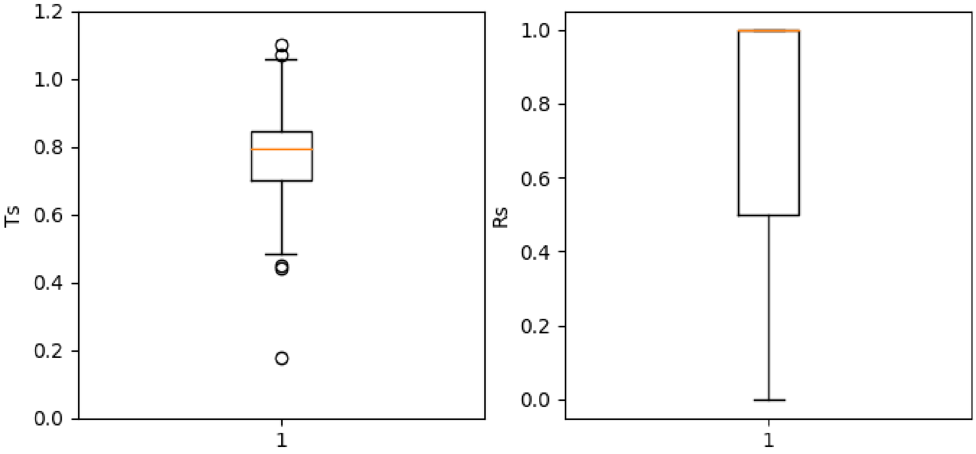
Figure 6.1 shows $T_s$ and $R_s$ for all methods in the sequence. In Figure 6.1(a), we see that in almost every case, Green speeds up the analysis (i.e., the time ratio is less than 1.0). In the few instances in which the time ratio is greater than 1.0, only one or two constraints need to be solved and the time difference of running SPF with and without Green is a matter of milliseconds.

Figure 6.1(b) shows the the reuse ratio $R_s$, which fluctuates between 0.0 and 1.0 with no apparent saturation as the number of programs increases. Figure 6.2 shows the minimum, maximum, median and upper/lower quartiles of $T_s$ and $R_s$. We see a median time ratio of 0.82 and a median reuse value of 1.0. We also notice a broad distribution of reuse values, ranging from 0.0 to 1.0.



(a)

Figure 6.1: Green reuse and time ratios of all methods.



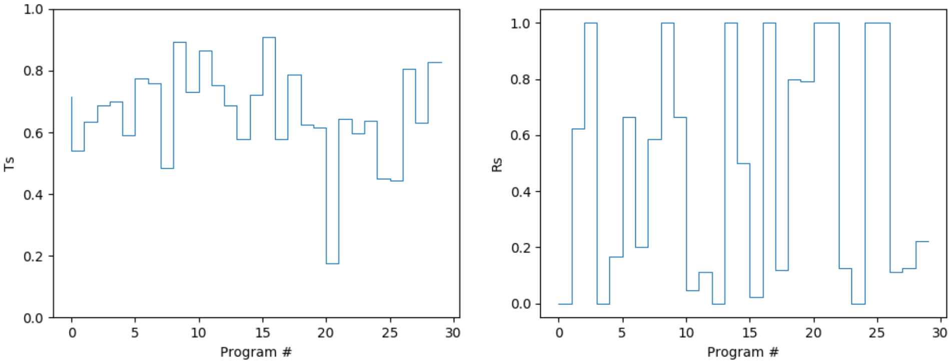
(a)

Figure 6.2: Boxplots of $T_s$ and $R_s$ for all methods.

Between the variables $T_s$ and $R_s$, we calculate a correlation coefficient of 0.01 with a p-value of 0.86, indicating a statistically insignificant correlation.

### Constraint-heavy Methods

Filtering the dataset for methods with 6 or more constraints yields 30 methods, the results of which are shown in Figures 6.3 and 6.4. We see in Figure 6.3(a) that Green speeds up the analysis in every case, with a median time ratio of 0.72 (see Figure 6.4(a)). In Figure 6.3(b), we see the reuse ratio fluctuating between 0.0 and 1.0, with a median reuse ratio of 0.36 (see Figure 6.4(b)).



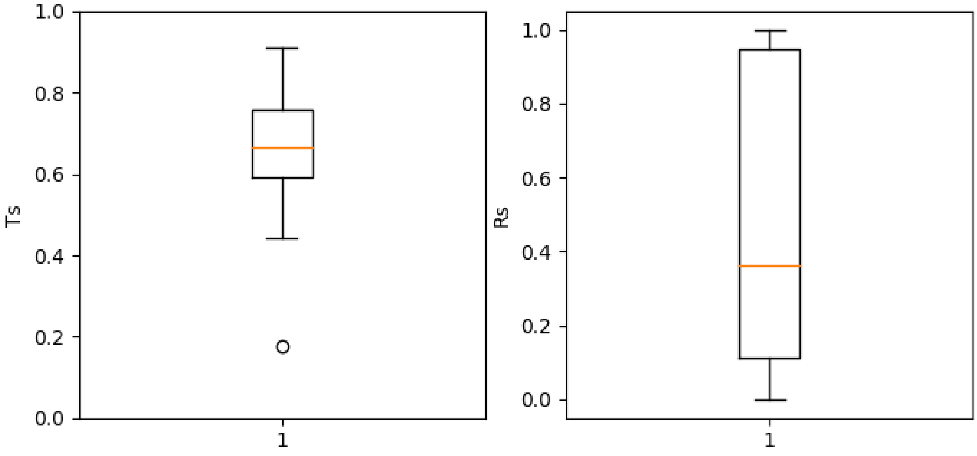
(a)

Figure 6.3: Green reuse and time ratios of constraint-heavy methods.

Between $T_s$ and $R_s$, we compute a correlation coefficient of -0.43 with a p-value of 0.02, indicating a negative correlation that is statistically significant.

## 6.2.4 Discussion

We notice several differences between the results of our analysis and those of the original study. When measuring the reuse that takes place across programs, the authors see the quartiles move towards 1.0 as more programs are analyzed. However, we do not see a similar trend in our experiment. The fraction of SAT queries reused consistently fluctuates between 0.0 and 1.0 as more programs are analyzed.



(a)

Figure 6.4: Boxplots of $T_s$ and $R_s$ for constraint-heavy methods.

The previous findings demonstrate that the SPF runtime decreases as more SAT queries are reused. This suggests a negative correlation between the variables $T_s$ and $R_s$. But, in our dataset of all methods, we see no significant correlation between $T_s$ and $R_s$. Yet, we do observe a statistically significant negative correlation in the dataset of constraint-heavy methods.

Although we do not always see a correlation between the reuse and time ratios, we do see a consistent decrease in the runtime when Green framework is employed. This drop in runtime is even more prevalent in the constraint-heavy methods, which actually has a lower reuse rate of 0.36. This could indicate the reason why the speed up of the analysis is independent from the reuse component. For example, Green may express constraints in a format that is beneficial to solvers, that is, canonization and normalizing may affect the runtime. The authors of [25] came up to similar conclusions, that solvers are sensitive to the format in which constraints are expressed.

While the authors use only six programs to perform their evaluation, we were able to use 94 programs consisting of 151 benchmark methods. Furthermore, the programs were taken from a variety of real-world applications and tailored specifically to SPF analysis. With this larger dataset, we were able to gain insightful information that would not have been possible from running the experiment with only a handful of programs. Therefore, the answer to our second research question is positive too: the study with a larger set of benchmark programs does arrive at the different conclusions.

**RQ2.** The answer to our second research question is positive: the larger set of benchmark programs can affect the evaluation.

### 6.2.5 Threats to Validity

#### Internal

As stated above, the slicer component of the Green solver framework did not appear to be working during our experiment. This could contribute to differences between our results and those of the original experiment. Additionally, the code we use to perform our evaluation may have bugs. To reduce this threat, we verify the output of SPF with and without the use of Green on select programs as we automate the process. We also use libraries (SciPy and Scikit-learn) for aggregating the data and collecting statistics.

#### External

The programs used in our experiment may not be representative. To reduce this threat, we use our tool to select a large number of programs from open-source repositories, which yields a wide-range of real-world programs with a varying number of parameters and conditional statements.

# CHAPTER 7

# CONCLUSIONS

This thesis presents the APT tool, a framework which automates the generation of realistic benchmark programs suitable for the symbolic execution tool SPF. This is done by extracting suitable programs from open-source repositories, then transforming them into compilable, stand-alone benchmarks by removing their external dependencies and nonlinear symbolic expressions. We designed the transformation algorithm around our concept of abstract-semantic program equivalence; that is, we apply transformations which guarantee semantic equivalence in the abstraction of symbolic analysis. In this way, we preserve program behaviors that are relevant to SPF, which allows us to produce a large set of realistic, non-trivial benchmark programs.

We implemented an instance of the APT tool to acquire a large number of benchmark programs for SPF, increasing by nine times the number of available benchmarks for this tool. To evaluate their impact, we replicate an evaluation of the Green solver framework and compare the results of our evaluation to those of the original study. Our results show that the increased number of benchmark programs does in fact affect the evaluation and hence the derived conclusions. Our case study demonstrates the importance of having a large, diverse set of program benchmarks available, which necessitates the means for automatically obtaining such benchmarks.

## 7.1 Future Work

The transformer component of the APT framework is part of a web-based Program Analysis Collaboratory (PAClab) tool [7] which uses user-defined selection criteria to obtain realistic benchmarks suitable for a specific verification task. Based on selection criteria, PAClab identifies relevant programs from open-source repositories, obtains those programs, and if necessary performs sound program transformations to adapt them to the verification task. PAClab makes the resulting program benchmarks available for download. PAClab is designed as a scalable, modular, and parametrizable tool that takes advantage of a computer cluster to handle multiple user requests.

Currently, the transformer component targets intra-procedural analyses with integer abstract domains. This component will be extended to support different analyses, for example, inter-procedural analyses where a benchmark consists of an entire Java class rather than a single method. The current implementation also only supports one evaluation environment, i.e., SPF, and will be extended to support different environments as well. Researchers will then be able to use PAClab's UI to select different filtering and transformation options that are relevant for their research. With this parameterization, the tool will be able to assemble various benchmarks for different verification environments and verification tasks. Program analysis researchers will then be able to obtain a variety of scoped program benchmarks, allowing them to focus on developing analysis techniques rather than searching for suitable artifacts.

# REFERENCES

[1] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Eliaand, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[3] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* $19^{th}$ *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[5] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.

[6] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 411–421, New York, NY, USA, 2013. ACM.

[7] Rebecca Brunner, Maria Paquin, Elena Sherman, and Robert Dyer. Paclab: a program analysis collaboratory. In *under submission*, TACAS 2020, 2020.

[8] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2008, pages 443–446, 2008.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. pages 209–224, 2008.

[10] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[11] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 49–60, New York, NY, USA, 2016. ACM.

[12] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, October 2005.

[13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[14] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

[15] Robert Dyer and Elena Sherman. Cri: Ci-p: Collaborative: Towards a program analysis collaboratory, August 2018. Award Abstract Number 1823294.

[16] Jean-Guillaume Fages and Jean-Guillaume Fages. Choco-solver. `https://github.com/chocoteam/choco-solver`, 2018.

[17] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[18] Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. Statistical symbolic execution with informed sampling. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 437–448, New York, NY, USA, 2014. ACM.

[19] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, 24(2):8:1–8:42, December 2014.

[20] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 581–591, New Delhi, 2015. Springer India.

[21] Hussain Ishtiaque, Csallner Christoph, Grechanik Mark, Xie Qing, Park Sangmin, Taneja Kunal, and Mainul Hossain B. M. Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software: Practice and Experience*, 46(3):405–431, October 2014.

[22] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 177–187, New York, NY, USA, 2015. ACM.

[23] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[24] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 554–559, New York, NY, USA, 2016. ACM.

[25] Justin Lloyd and Elena Sherman. Minimizing the size of path conditions using convex polyhedra abstract domain. *SIGSOFT Software Engineering Notes*, 40(1):1–5, February 2015.

[26] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 12 2017.

[27] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in java projects: An empirical study. In *Proceedings of the*

*13th International Conference on Mining Software Repositories*, MSR '16, pages 500–503, New York, NY, USA, 2016. ACM.

[28] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, first edition, 1999.

[29] Peter Ohmann, Alexander Brooks, Loris D'Antoni, and Ben Liblit. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 390–405, New York, NY, USA, 2017. ACM.

[30] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 35:1–35:11, New York, NY, USA, 2012. ACM.

[31] Michael Y. Levin Patrice Godefroid and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Symposium on Network and Distributed System Security*, 2008.

[32] G.D. Plotkin. Call-by-name, call-by-value and the ?-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[33] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 34–44, New York, NY, USA, 2011. ACM.

[34] Corina S. Păsăreanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*. Springer-Verlag, 2004.

[35] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[36] Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Compositional symbolic execution with memoized replay. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 632–642, Piscataway, NJ, USA, 2015. IEEE Press.

[37] Vinayak Sinha, Alina Lazar, and Bonita Sharif. Analyzing developer sentiment in commit logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 520–523, New York, NY, USA, 2016. ACM.

[38] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[39] Tian Tan, Yue Li, and Jingling Xue. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 278–291, New York, NY, USA, 2017. ACM.

[40] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[41] Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 263–277, New York, NY, USA, 2017. ACM.

[42] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, and Denys Poshyvanyk. License usage and changes: A large-scale study of java projects on github. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 218–228, Piscataway, NJ, USA, 2015. IEEE Press.

[43] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[44] Guillaume Brat SeungJoon Park Willem Visser, Klaus Havelund and Flavio Lerda. Model checking programs. In *Automated Software Engineering*, ASE '10, pages 203–232. Kluwer Academic Publishers, 2003.

[45] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM.

[46] Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 344–358, New York, NY, USA, 2017. ACM.