

**IMPROVING IN-HOUSE TESTING  
USING FIELD EXECUTION DATA**

A Dissertation  
Presented to  
The Academic Faculty

By

Qianqian Wang

In Partial Fulfillment  
of the Requirements for the Degree  
Doctor of Philosophy in the  
School of Computer Science

Georgia Institute of Technology

December 2019

Copyright © Qianqian Wang 2019

**IMPROVING IN-HOUSE TESTING  
USING FIELD EXECUTION DATA**

Approved by:

Dr. Alessandro Orso, Advisor  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Vivek Sarkar  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Spencer Rugaber  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Qirun Zhang  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Yuriy Brun  
College of Information & Computer Science  
*University of Massachusetts,  
Amherst*

Date Approved: August 01, 2019

To my husband, Jake Cobb, for his unconditional support.

## ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Dr. Alessandro Orso. Without him, I would not have been able to finish my doctorate degree. He guided me through the process of finding research ideas, building research projects and writing papers that lead to my thesis dissertation.

I also want to thank my committee members. They provided insightful comments and suggestions for improving this dissertation.

I would also like to thank my collaborators and labmates. They generously shared their valuable opinions and experience in our daily interactions and discussions, which helped me face challenges during my Ph.D. study.

My research would not have been the same without students who contributed to my user studies and provided helpful experiment data. I appreciate their participation.

I also want to thank my family. They supported and encouraged me throughout my study.

Last but not the least, I would like to give my special thanks to Dr. Mary Jean Harrold, who gave me the opportunity to start my Ph.D. and inspired me to carry on through my Ph.D. years. Her spirit continues to guide me in the rest of my life.

## TABLE OF CONTENTS

<b>Acknowledgments</b> . . . . .	v
<b>List of Tables</b> . . . . .	x
<b>List of Figures</b> . . . . .	xi
<b>Chapter 1: Introduction and Motivation</b> . . . . .	1
1.1 Thesis Statement . . . . .	2
1.2 Approaches . . . . .	2
1.3 Contributions . . . . .	5
<b>Chapter 2: Background</b> . . . . .	7
2.1 Mutation Testing . . . . .	7
2.2 Temporal Invariants . . . . .	9
2.3 Symbolic Execution and Guided Symbolic Execution . . . . .	11
2.4 Terminology . . . . .	12
<b>Chapter 3: Overall Vision</b> . . . . .	14
<b>Chapter 4: Behavioral Execution Comparison</b> . . . . .	17
4.1 Challenges in Comparing In-House vs. Field Executions . . . . .	17
4.2 Behavioral Models for Difference Measuring . . . . .	19

4.2.1	Behavioral models . . . . .	19
4.2.2	Behavioral comparison metrics . . . . .	21
4.3	Experiment Methodology . . . . .	21
4.3.1	Software Benchmarks . . . . .	23
4.3.2	Computing Behavioral Comparisons . . . . .	26
4.3.3	Publicly Released Dataset . . . . .	26
4.4	Results and Discussion . . . . .	27
<b>Chapter 5: Mimicking User Behavior Using Field Execution Data . . . . .</b>		<b>36</b>
5.1	Motivating Example . . . . .	36
5.2	An Approach for Mimicking Executions . . . . .	38
5.2.1	Overview . . . . .	38
5.2.2	Execution Instrumenter and Processor . . . . .	39
5.2.3	Execution Mimicker . . . . .	40
5.2.4	Applying Replica to the Motivating Example . . . . .	46
5.3	Evaluation . . . . .	47
5.3.1	Replica Implementation . . . . .	48
5.3.2	Program Considered and Field Data Collection . . . . .	48
5.3.3	Experiment Protocol . . . . .	50
5.3.4	Results and Discussion . . . . .	51
5.3.5	Limitations and Threats to Validity . . . . .	55
<b>Chapter 6: Further improvement on test generation . . . . .</b>		<b>57</b>
6.1	Motivation . . . . .	57

6.2	Intuition Validation . . . . .	58
6.2.1	The Impact of Variable Replacement . . . . .	59
6.2.2	Limited-Valued Variables . . . . .	61
6.3	An Approach for Improving Symbolic Execution Using Observed Values . . . . .	63
6.3.1	Overview . . . . .	64
6.3.2	Instrumenter . . . . .	64
6.3.3	Value Analyzer . . . . .	65
6.3.4	Input Generator . . . . .	66
6.4	Evaluation . . . . .	70
6.4.1	Implementation . . . . .	70
6.4.2	Subject Programs . . . . .	71
6.4.3	Experiment Setup . . . . .	72
6.4.4	Results and Discussion . . . . .	73
6.4.5	Limitations and Threats to Validity . . . . .	77
<b>Chapter 7: Related Work . . . . .</b>		<b>78</b>
7.1	Using Field Data in Software Engineering . . . . .	78
7.2	Behavioral Representation . . . . .	79
7.3	Test Data Generation . . . . .	81
7.4	Reproducing Field Failures . . . . .	82
7.5	Improving Constraint Solving for Symbolic Execution . . . . .	83
<b>Chapter 8: Conclusion and Open Problems . . . . .</b>		<b>84</b>

**References** . . . . . 95

## LIST OF TABLES

2.1	A list of common mutation operators (Adapted from [17]). . . . .	8
2.2	Example of killed mutants and mutation score. . . . .	9
4.1	Study benchmarks. . . . .	22
4.2	Comparison between developer-written tests and field executions. . . . .	29
4.3	Augmented tests. . . . .	31
4.4	Comparison between augmented tests and field executions. . . . .	32
5.1	Subject programs used in the study. . . . .	49
5.2	Experiment results. . . . .	51
5.3	Mutants killed by individual replicas. . . . .	54
6.1	SMT-LIB benchmarks. . . . .	59
6.2	Benchmarks used to evaluate VASE. . . . .	71
6.3	Constraint Solving Performance. . . . .	73
6.4	Percentage of improved executions . . . . .	75

## LIST OF FIGURES

2.1	A simple program and its test suites. . . . .	8
2.2	Two example Commons IO execution traces and the kTails invariants in those traces. . . . .	10
2.3	Simple code example to illustrate guided symbolic execution. . . . .	12
3.1	Overall vision of my thesis. . . . .	15
3.2	High-level overview of my techniques. . . . .	16
4.1	Overview of the experiment setup: field data collection, model inference, and execution comparison. . . . .	25
5.1	A motivating example. . . . .	37
5.2	High-level view of Replica. . . . .	39
5.3	Execution Mimicker. . . . .	41
5.4	Mutants killed by individual replicas. . . . .	55
6.1	Performance change after variable replacement. . . . .	60
6.2	Simple code example to limited-valued variables. . . . .	61
6.3	High-level overview of VASE. . . . .	64
6.4	Input Generator. . . . .	67
6.5	Coverage Improvement . . . . .	76

## SUMMARY

As software permeates people’s everyday life and work, software quality assurance becomes more and more important. In-house testing, despite being the most widely used approach for assessing and improving software quality, has its inherent limitations. First, tests written in-house tend to be incomplete. Exhaustive testing is infeasible even for simple programs due to the infinite number of possible paths as well as time and resource limitations. Therefore, developers are forced to choose only a subset of program structure and behavior to test. Second, testing can be inaccurate because it is usually performed by software developers or testers, who make assumptions on how the software will be used by real users. These assumptions can be inaccurate, or even incorrect, creating a gap between how the software is tested in-house and how it is used in the field. As a result of the two limitations above, in-house tests are typically not representative of the software behavior exercised in the field.

To assess and mitigate the shortcomings of in-house tests, I performed three main pieces of research. I will now provide an overview of these three parts of my dissertation.

In the first part, I defined four models—two coverage-based, one mutation-based and one temporal-invariant-based— to represent software execution behavior. I then collected field execution information for four real-world systems and measured the degree to which in-house tests are representative of real field executions in terms of each defined model. The study results suggest that even high-coverage test suites miss a considerable portion of software behavior exercised in the field.

In the second part, I defined Replica, a test generation technique that can generate new test inputs and augment in-house test suites by mimicking observed user behavior. Replica (1) collects lightweight field execution data, represented as method call sequences; (2) identifies behavior exercised in the field but missed by in-house tests; and (3) generates test cases that exhibit the missed behavior using an incremental, guided symbolic execution

approach.

Finally, I defined Value-Aided Symbolic Execution (VASE), a technique that further improves the effectiveness of symbolic test generation. With VASE, I used observed program states to improve the performance of constraint solving, an inherently expensive operation in symbolic execution. To do so, I identified variables, at specified program points, that (1) have limited number of observed values across executions, or (2) exhibit properties for all the observed values. I then leveraged these observed values or properties to provide extra information to constraints generated during symbolic execution.

I implemented my techniques in prototype tools and evaluated them on real-world programs using realistic data collected from real users. The evaluation shows that my techniques can effectively generate test inputs using field execution data and make in-house testing more representative of field executions.

## **CHAPTER 1**

### **INTRODUCTION AND MOTIVATION**

As software becomes prevalent in our daily life, software quality assurance gets more and more important. In the meanwhile, software testing becomes more challenging than ever before. First, the software itself is getting more complicated and it is impossible to test it thoroughly. Second, software is deployed on millions of machines with different environment and testing may have limited access to evaluate the software in all the possible environment. Last but not least, software is used by millions of users; they use the software in different ways out of different habits or understanding of the software. All the above reasons make it complicated for tester-written tests to be representative of real uses performed by users in the field. As a result, a large number of bugs may be hidden until the software is released and used by real users.

Therefore, to address the shortcomings of in-house tests, it is necessary to identify and understand in detail the differences between in-house tests and field executions. However, there are no well-understood ways of measuring differences between them, due to two main challenges. First, it is hard to establish standards for execution representation that can be used for the behavioral comparison between different sets of executions. Various model inference techniques have been proposed to characterize a set of executions (e.g., [1, 2, 3, 4, 5]), but these techniques have not been studied for their usefulness in identifying differences between sets of executions. In fact, it is difficult to filter out trivial differences while keeping all the important ones. Second, it is difficult to collect field data to build these models. To make the execution comparison representative, field data has to be collected from a large number of users who uses the software in a real-world setting. Without proper ways to collect proper types of information, the data collection process may pose privacy concerns and affect the users' experience due to computational overhead.

In addition to identifying the gap between in-house testing and field executions, the more important step is to bridge the gap. Researchers and practitioners proposed various techniques for complementing in-house test suites. Beta testing [6] and staged rollout (e.g., [7]), for instance, are commonly used approach that allows a subset of the user population to use yet-to-be released software, so that developers can gather information on field usage before wider release. Perpetual testing [8] proposes to continue testing and analysis in deployment phase and throughout the lifeline of the application. Many techniques have also been proposed by researchers for automatically generating test data to complement manual testing. For example, Randoop [9] uses a feedback-directed approach to randomly generate test inputs. Klee [10] applies a symbolic-execution-based technique for test generation. EvoSuite [11] automatically generates unit test cases for object-oriented programs using hybrid search algorithms. Although these techniques can improve the situation, we are still witnessing a gap between in-house tests and field usage.

## **1.1 Thesis Statement**

The thesis statement of the dissertation is that automated test generation approaches that leverage information collected from field uses and represented with appropriate models can bridge the gap between in-house tests and field executions and improve the quality of the test suites.

## **1.2 Approaches**

To confirm my thesis statement, I (1) performed an in-depth study, (2) defined two test generation techniques, Replica and VASE, and (3) evaluated them on real world programs. In the rest of this section, I present an overview of the study and the techniques.

As a first step towards my goal of the thesis statement, in Chapter 4, I designed a study that aims to understand in-depth (1) whether differences exist between in-house tests and field executions, (2) if so, to what degree they differ from each other and, (3) what

representation can be used to capture these differences.

More precisely, I studied four real-world software systems that represent two common ways software can be used: by end-users or by other (client) software. All four systems come with in-house test suites that achieve over 75% code coverage. I collected end-user executions for one system and executions through client code for the other three. To compare the behavior of in-house tests and field executions, I used four models based on different metrics: two coverage-based models (statement and method coverage), one mutation-based model (killed mutants [12]) and one temporal-invariant-based model (kTails-based invariant [13, 2]). In addition, I investigated whether existing automatic test generation techniques could help improve the representativeness of in-house tests. Specifically, I augmented developer-written tests using a state-of-art test generation tool and analyzed the difference between the augmented test suites and field executions.

There are three major findings. First, even high-quality test suites miss a large portion of software behavior exercised in the field. Second, existing test generation techniques can only marginally reduce the missed behavior. Third, the invariant-based model is more effective in identifying differences between in-house tests and field executions.

To address the problem of low representativeness of in-house tests, in Chapter 5, I defined a technique called Replica, which automatically generates new test cases that can mimic observed user behavior. To do so, and unlike most existing automated test generation techniques, Replica incorporates lightweight field execution information into the process of test generation.

Briefly, Replica works as follows: It takes as input a program  $P$  and a (in-house) test suite  $T$  for  $P$  and works as follows. First, Replica runs an instrumented version of  $P$  against  $T$  and identifies which behaviors  $T$  exercises, in the form of a set of dynamic invariants  $D$ . Second, it generates another instrumented version of  $P$  that, when executed in the field, collects execution data and checks whether untested behaviors are exercised (i.e., whether the execution violates one or more invariants in  $D$ ). (The execution data collected by the

current instance of Replica are method call sequences, which can be collected with low overhead.) If there is a violation, the execution data are stored, together with information on the location of the violation(s). Finally, Replica generates new tests for P that exercise the same new behaviors observed in the field using an incremental, guided symbolic execution approach. Intuitively, Replica attempts to generate a test that violates the same invariant(s) as the observed field execution by first trying to reach the invariant-violating point(s) in P directly, and then incrementally adding intermediate points in the collected execution data to reduce the search space. This incremental approach allows Replica to explore the execution space without over-constraining the search and by adding additional constraints as needed. It also allows, when successful, for generating test inputs that are potentially quite different from the user inputs, thus mitigating privacy-violation concerns.

To assess the usefulness of Replica, I performed an empirical evaluation in which I implemented Replica for C program and applied it to a real-world program, its developer-provided in-house test suite, and a set of execution data collected from 747 real users. First, I assessed the effectiveness of Replica by measuring how many of the untested behaviors (i.e., invariant violations) exercised by the real users were also exercised by the new tests generated by the approach. The results are encouraging, in that Replica was able to successfully generate test inputs that exercised 73.7% of the new behaviors observed in the field. I then assessed the fault detection capability of the newly generated tests by creating over 400 mutants of the program and measuring how many were killed by the original in-house test suite, the field executions, and the new tests. Also in this case, the results of the evaluation are promising: the newly generated tests were able to detect (i.e., kill) 79.4% of the mutants detected by the field executions but not by the in-house, developer-provided tests.

Despite Replica shows effectiveness in mimicking user behavior, it still has some limitations that are inherited from symbolic execution, among which the cost of constraint solving is a key bottleneck. To mitigate this issue, I defined a technique, VASE, in Chapter 6

that aims to improve the effectiveness and efficiency of symbolic execution by leveraging observed values from user executions. VASE first instruments programs at each branch statement such that the instrumented program can record the concrete values observed at that point. Then, it analyzes the recorded values to (1) identify variables that have only limited number of observed values, or (2) infer invariants that hold for all the values that are observed at the same program points. Next, it performs classic symbolic execution with the following modifications. (1) Whenever the execution needs to consult the underlying solver, VASE first checks whether there is any invariant inferred at this program point. If so, it replaces the symbolic variable with an observed value or adds additional information to the constraint. Finally, with the replaced variable or added information, VASE simplifies the constraint before passing it to the underlying solver.

To evaluate the effectiveness of VASE, I implemented a prototype on top of the Symbolic PathFinder [14], evaluated it on four real world programs and compared the results with Green [15], a state-of-art symbolic execution tool that aims to reduce the cost of constraint solving. The results show that VASE can reduce the time used for solving each constraint by 32%. It can also reduce the number of timeout constraints by 57%. By doing so, it increases the number of reused constraints and reduces the number of solver invocations by 54%. Finally, since the time used in the solver is reduced, VASE has time to explore more execution path and covers more instructions than Green. These results show that VASE is effective in reducing the cost of constraint solving and improving code coverage.

### **1.3 Contributions**

This thesis dissertation provides the following novel contributions:

- A study of the differences between field executions, in-house test suites, and test suites augmented with automatically generated tests on four real-world systems.
- A methodology and a set of models for evaluating the behavioral differences between

executions, including the use of temporal invariants for detecting fine-grained differences.

- A technique based on incremental, guided symbolic execution that can generate tests that mimic observed user behavior.
- A novel technique that uses observed values to reduce the cost of constraint solving in symbolic execution.
- The implementation of all techniques in two prototype tools.
- An empirical evaluation of the technique performed on a real program, real tests, and real user executions.
- A publicly available dataset of test and field executions for four real-world systems that can be used to replicate and advance my study results.
- A dataset that contains my evaluation artifacts, including execution data collected from real users.

The rest of the thesis dissertation is organized as follows. Chapter 2 provides some necessary background information and defines some relevant terminology. Chapter 3 introduces my overall vision of achieving my overarching goals. Chapter 4 describes the details of my experiment methodology and results for the first study, and provides possible research implications. Chapter 5 describes the details of my test generation approach, Replica, and presents the empirical evaluation. Chapter 6 describes the technique VASE and discusses the evaluation results. Chapter 7 discusses related work. Finally, Chapter 8 concludes the thesis dissertation and discusses some future problems related to the dissertation.

## CHAPTER 2

### BACKGROUND

Before discussing my approaches, I briefly provide some necessary background information on mutation testing, temporal invariants and symbolic execution. I will then define some terms that I use in the rest of my thesis dissertation.

#### 2.1 Mutation Testing

Mutation testing [16] is an approach for measuring the quality of test suites. It introduces minor changes to a program, one at each time, and generates multiple versions of the program. Each mutated version is called a mutant and they are used to assess whether the test suites can detect the change by causing different behavior from the original version of the program. If a change is detected by test suites, we say the mutant containing the change is killed. The percentage of mutants that a test suite kills is called mutation score and it is used to measure a test suite's fault revealing ability.

Mutation operators are transformation rules that define how to modify the original program to generate mutants. They involve modifying variables or expressions by replacement, insertion or deletion. The most common mutation operations are listed in Table 2.1 [17].

I use a simple program to show how mutation operators are applied. Figure 2.1 shows a simple program that computes the sum of two integers, along with three sets of test inputs ( $ts1$ ,  $ts2$ ,  $ts3$ ) to the program. Each test suite contains three test cases and each test case is represented as  $(a, b)$ . For example, the first test case in  $ts1$  means  $a = 1$  and  $b = -1$ .

I apply the AOR (arithmetic operator replacement) operator on statement 1 and get four mutants: each mutant differs from the original program in that the expression  $a + b$  is changed to  $a - b$ ,  $a * b$ ,  $a/b$  and  $a\%b$ , respectively. Table 2.2 demonstrates whether each test suite kills the generated mutants. Each row refers to a mutant and each column refers to

Table 2.1: A list of common mutation operators (Adapted from [17]).

Mutation operators	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

a test suite. The symbol **Y** in a cell means the test suite in the column kills the mutant in the corresponding row and **N** means otherwise. Finally, the mutation score for each test suite is calculated based on the ratio of mutants it kills to the total number of mutants generated.

```
[tabsize=4]
int sum(int a, int b) {
1. return a + b;
}
|
| ts1: (1, -1) (1, 1) (2, 3)
| ts2: (0, 1) (1, 0) (-1, 2)
| ts3: (2, 2) (-1, 1) (1, 0)
```

Figure 2.1: A simple program and its test suites.

Table 2.2: Example of killed mutants and mutation score.

	ts1	ts2	ts3
$a + b \rightarrow a - b$	N	Y	Y
$a + b \rightarrow a * b$	N	N	Y
$a + b \rightarrow a / b$	Y	Y	Y
$a + b \rightarrow a \% b$	Y	Y	Y
Mutation score	0.5	0.75	1

## 2.2 Temporal Invariants

Invariants are properties that can be relied upon to be true during the execution of a program, or during some portion of it. Invariants of software systems have been shown useful to capture and understand software behavior (e.g., [18, 19, 20, 21, 22]). Temporal invariants [23, 24, 25] are properties that constrain the order in which program events take place. For example, opening a file should always happen before reading from or writing to the file and it should eventually be followed by closing the file.

Various temporal invariants can be used to model program behavior given the program’s execution traces. In a case study of hundreds of properties, Dwyer et al [25] developed a set of invariants that are the most frequently observed. More specifically,  $a \rightarrow b$  (*a Always Followed by b*) means whenever the event  $a$  occurs, the event  $b$  must occur late in the same trace.  $a \leftarrow b$  (*a Always Precede b*) refers to the property that whenever  $b$  occurs,  $a$  must have already occurred before  $b$ .  $a \nrightarrow b$  (*a Never Followed by b*) denotes the invariant that whenever  $a$  occurs,  $b$  never occurs later in the same trace.

Serving as a basis for the above temporal invariants and other behavioral model inference algorithms (e.g., [26, 27, 28, 29, 30, 31, 32, 5]), kTails has been shown to (1) infer precise models [5] and (2) scale to large sets of executions [2].

The inputs of the kTails algorithm are an integer  $k$  and a set of execution traces, each a totally-ordered set of events that take place during one execution. The algorithm represents each trace as a linear finite state machine and iteratively merges the events that are followed

```
Trace A: read, readFirstBytes, getBOM, close  
Trace B: read, readFirstBytes, getBOM, length
```

**kTails invariants:**

```
getBOM → length → END  
getBOM → close → END  
readFirstBytes → getBOM → length | close  
close → END  
read → readFirstBytes  
length → END  
readFirstBytes → getBOM  
read → readFirstBytes → getBOM  
getBOM → length | close  
START → read  
START → read → readFirstBytes
```

Figure 2.2: Two example Commons IO execution traces and the kTails invariants in those traces.

by the same sequence of up to length  $k$ . A recent formulation of the kTails algorithm, *InvariMint*, demonstrated that the final FSM can be uniquely described by the set of specific types of temporal invariants mined from the execution traces [2, 3]; I use this formulation in my work. That is, I use *InvariMint* to mine temporal kTails invariants from the field and from test execution logs and compare those sets of invariants. For my experiments, I found that using  $k = 2$  worked well in practice, whereas larger  $k$  caused *InvariMint* to run out of memory for some long traces. Models using larger  $k$  values would capture more execution data, but I leave for future work the investigation of whether the extra data captures useful behavioral information that would otherwise be missed.

Fig. 2.2 shows two examples of simplified traces from Commons IO executions and the kTails invariants ( $k = 2$ ) mined from them.  $\rightarrow$  means “can be immediately followed by”, and each invariant can have up to  $k \rightarrow s$  ( $k = 2$  in my case).  $|$  means “or”, and I use it as shorthand to display multiple invariants. For example, the line `readFirstBytes → getBOM → length | close` represents two invariants: one that says that methods `readFirstBytes`, `getBOM`, and `length` occurred consecutively in an execution; and another that says the methods `readFirstBytes`, `getBOM`, and `close` did as well.

### 2.3 Symbolic Execution and Guided Symbolic Execution

Symbolic execution [33] is an approach for analyzing a program to determine what inputs cause each part of the program to execute. It executes the program with symbolic instead of concrete inputs. The execution maintains a *symbolic state* and a set of constraints, called *path conditions*(PC), at each program point. Each state is a map between a set of memory addresses to a set of possible symbolic values, represented as expressions built from symbolic inputs to the program.

The symbolic execution engine initializes the path condition to *true* and builds the symbolic states and path conditions incrementally as the execution proceeds. The symbolic state is updated when a statement that modifies the value of a memory location is executed. When a branch statement is executed, symbolic execution consults whether the condition in the predicate can be *true* or *false* given the existing path condition, indicating which branch the execution can take. If both branches are feasible, symbolic execution forks and follows both branches; otherwise it only follows the feasible branch. In the meanwhile it updates path conditions by adding a conjunct that represents predicate being *true* or *false*, depending on the branch. When a termination criterion is met, e.g., encountering exit point of the program or user-defined termination point, symbolic execution asks a solver for a solution that satisfies the path condition. The solution assigns concrete values to the symbolic variables and thus can be used as inputs that cause the program to execute a particular path. Traditional symbolic execution is commonly used to explore as many paths as possible in order to generate test inputs that can achieve high code coverage.

Guided symbolic execution, on the other hand, does not aim to generate high-coverage tests; instead, its goal is to generate test inputs that can cover particular program points that we are interested in. Guided symbolic execution takes in a program  $P$  and a set of target points  $T = \{t_0, t_1, \dots, t_n\}$ . It starts from the program entry with symbolic inputs and finds a path from the entry to  $t_0$  and checks the satisfiability of the corresponding path condition. If

the condition cannot be satisfied, a different path is chosen from the entry to  $t_0$ . Otherwise it repeats the process of finding feasible paths from  $t_i$  to  $t_{i+1}$  until it reaches  $t_n$ , when it asks the solver for a solution to the path condition and uses the solution as the inputs needed.

```
void foo(int a, int b, int c) {
1. if (a > b) {
2.   d = b + c
3.   if (d > a)
4.     //do something
5.   else
6.     //do something
7. }
8. if (a - 5 > b)
9.   if (c < 10)
10.    //do something
11. else
12.   //do something
13. return
}
```

Figure 2.3: Simple code example to illustrate guided symbolic execution.

To illustrate how guided symbolic execution works, I use a simple code example in Figure 2.3. The function *foo* takes in three integers  $a$ ,  $b$  and  $c$  as inputs and performs different operations depending on the inputs. Symbolic execution starts with symbolic values for  $a, b, c$ , denoted as  $a_s, b_s, c_s$ . If symbolic execution follows the path  $\langle 1, 2, 3, 6, 8, 9, 10, 13 \rangle$ , the symbolic state at the exit statement (line 13) is  $a = a_s, b = b_s, c = c_s$  and the generated path condition is  $a_s > b_s \wedge b_s + c_s \leq a_s \wedge a_s - 5 > b_s \wedge c_s < 10$ . At this point, an underlying solver is invoked to solve the path condition and it returns a satisfying solution (e.g.,  $a = 6, b = 0, c = 0$ ). This returned solution is the input tuple that causes the execution to follow the path  $\langle 1, 2, 3, 6, 8, 9, 10, 13 \rangle$ .

## 2.4 Terminology

In this section, I will define some terms that I am going to use in the rest of my thesis dissertation.

A *control flow graph (CFG)* for a function is a directed graph. Each node in the graph represents a basic block in the function and each edge represents a syntactically possible control flow between nodes.

An *interprocedural control flow graph (ICFG)* is a directed graph constructed by connecting a set of CFGs. If a function  $f_1$  calls a function  $f_2$ , their corresponding CFGs  $G_1$  and  $G_2$  are connected based on the following rules: the node  $n$  in  $G_1$  where the method call happens is split into two nodes  $n_c$  (call node) and  $n_r$  (return node), such that  $n_c$  succeeds all predecessors of  $n$  and  $n_r$  precedes all successors of  $n$ .  $n_c$  is connected to the entry node of  $G_2$  and the exit node of  $G_2$  is connected to  $n_r$ .

A *test trace* or a *field trace* is a sequence of method calls, distinguished by call site, invoked in order during testing or a field execution, respectively. An *invariant* is a subsequence observed in at least one *test trace*. An *invariant-violating sequence* is a subsequence of a *field trace* which is never observed in *test traces*. The method calls in an *invariant-violating sequence* are called *invariant-violating points* in the trace. They are used as *targets* to guide symbolic execution.

*Replica* refers to my general technique that uses incremental, guided symbolic execution for test generation. A test case generated by Replica is also called a *replica*. Each *replica* is generated by following a trace that represents a field execution and the field execution is called the replica's *original*.

## CHAPTER 3

### OVERALL VISION

In this chapter, I am going to provide my overall vision as well as the high-level overview of my approaches for achieving my overarching goals in my thesis statement and addressing why it is important to identify and bridge the gap between in-house tests and field executions.

Figure 3.1 provides the vision for my dissertation. In a nutshell, my research compares in-house tests and field executions for a given program and leverages the difference to generate test inputs that can be used to improve the quality of in-house testing. Figure 3.2 illustrates an overview of my approaches for achieving the vision.

Figure 3.2 provides a high-level overall vision of using field execution data to improve in-house test suites. As the figure shows, there are three main processes in the overall vision.

The first process is *Instrumentation*, which takes in an application as input and outputs an instrumented application. This process inserts probes into the application so that it collects execution data when the application is run by either testers or field users. To understand the execution differences from different aspects, I collected several different types of execution data to build various models (discussed in Chapter 4) and used the most expressive model as guidance for mimicking user behaviors (Chapter 5). I chose different techniques and tools for instrumentation, based on the distinct characteristics of applications I investigated. The detailed implementation of instrumentation process can be found in the experiment methodology section of Chapter 4 and the approach section of Chapter 5.

The second process is *Comparison*, which is the core approach to achieve my first goal of identifying important differences between in-house tests and field executions. In this process, my approach takes in the execution data collected from both test and field executions and outputs the differences between the two sets of executions. I will define

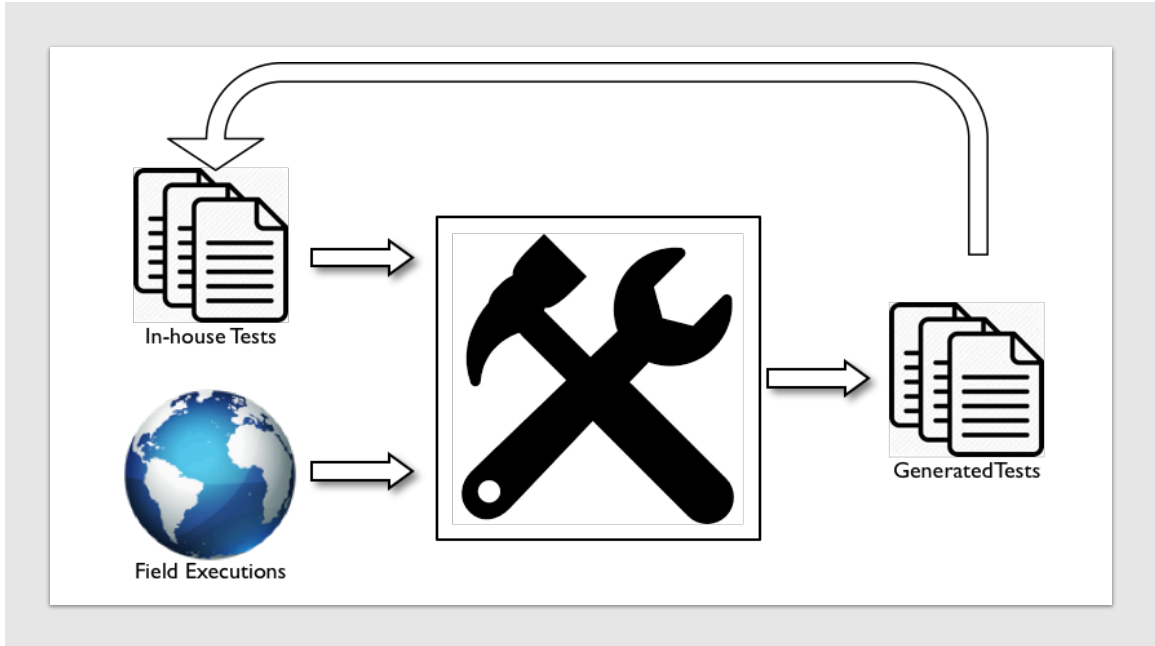


Figure 3.1: Overall vision of my thesis.

“execution differences” in terms of different models, based on the types of execution data collected when the application was used. I will then present the experiment results in two dimensions: first analyze the differences between in-house tests and field executions and then compare the behavioral models in terms of their effectiveness in identifying differences.

The third process is *Test Generation*, which is the most challenging component of my research. In this process I present two techniques that uses field execution data in two aspects. My first technique takes the application and field execution data as inputs. The field execution data is analyzed in the second process and contains information about the identified difference between in-house tests and field executions. The expected output of the process is one or more executions that exhibits the “same behavior” only observed in the field. I will define the “same behavior” when I describe the detailed technique in Chapter 5. Specifically, I used an incremental, guided symbolic execution algorithm that allows the generated test cases to be different from the original field execution while leading the execution to cover the same untested field behavior. My second technique further improves the efficiency of test generation by reducing the cost of constraint solving. The basic idea

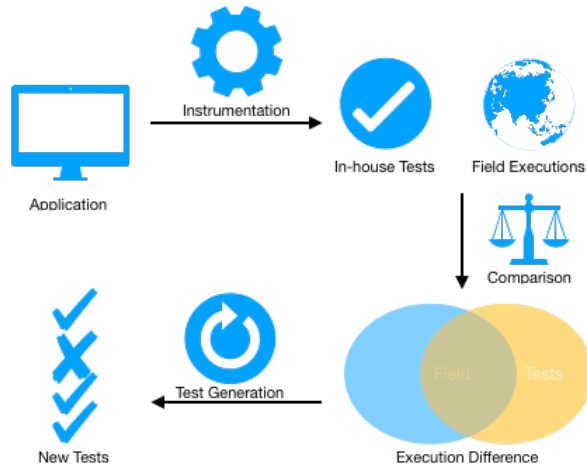


Figure 3.2: High-level overview of my techniques.

is to use values observed from user executions to provide additional information to the constraint solver. To do so, I collected variable values from concrete executions and inferred properties from them. During the symbolic execution, the inferred properties are added accordingly to the constraint to be solved. The details of this technique are presented in Chapter 6.

In summary, I present a general overall vision to address the problem of low representativeness of in-house tests. The overall vision are broken down to two goals: identifying the differences between in-house tests and field executions and reducing such differences with the help of user execution data. I will present my techniques and empirical studies that assess these techniques in the following chapters.

## **CHAPTER 4**

### **BEHAVIORAL EXECUTION COMPARISON**

In this chapter, I present in detail the study I performed on the differences between in-house tests and field executions. I start with the challenges face in the study and my solution to these challenges. I then describe the models I used to represent execution behaviors and the metrics I developed for comparing executions. After that I present the experiment methodology and results analysis.

#### **4.1 Challenges in Comparing In-House vs. Field Executions**

The goal of in-house testing is to validate that software executions adhere to an expected behavior. But the behavior tests validate may differ from the behavior exercised in the field.

While testing is the most widely used approach for assessing and improving software quality [34], it is subject to the developers' assumptions about the users' environments and behavior. These assumptions are necessary because non-trivial software cannot be tested exhaustively, and because of the vast diversity of the environments in which the software may execute, in terms of the underlying hardware, the operating system and its configuration, co-executing software, and so on. Even if it were possible to test the software in the multitude of environments representative of the users' environments, developers cannot predict all the ways in which users will use the software. In fact, developers and independent testers often envision the software is used in a prescribed manner, and these assumptions restrict the space of possible behavior considered. Beta testing aims to use real users to better approximate field executions, but beta testing is not automated, not repeatable, and only demonstrates a small slice of real-world field executions. As a result, released software typically contains bugs [35, 36, 37], most of which are unknown to the developers and do not have tests that expose them. In fact, the existence of behavioral differences between field executions and

in-house tests implies that software ships with untested behavior, and thus likely unknown bugs. Thus, the quality of the software cannot be properly assessed with existing test suites, and resources cannot be properly assigned to maintenance tasks. My work aims to identify and analyze the differences between in-house test suites and field executions from the point of view of behavior missed by the tests. This identification and analysis can provide insight into the behavior commonly missed by tests and help bridge the gap between tests and field executions, ultimately improving the effectiveness of testing.

The magnitude of the differences between tests and field executions has not been studied in detail, and there are no well understood ways of measuring these differences. While many development organizations collect data on the way their applications are used in the field, to the best of my knowledge no prior study has attempted to analyze and understand the differences between tests and field executions in a systematic way. Accordingly, my goals include developing a methodology for comparing executions' behavior and applying it to tests and field executions. There are two main challenges to my work:

**1. There are no established standards for comparing the behavior of sets of executions.** While executions can be characterized in various ways, and significant work on behavioral model inference has tackled the problem of summarizing a set of executions (e.g., [13, 26, 27, 28, 29, 30, 31, 38, 32, 39, 3, 2, 4]), this work has not considered directly comparing sets of executions. No behavioral execution representation is perfect for accurately representing all characteristics of that execution. For example, representing executions concretely with the complete trace of all executed instructions may expose trivial differences that are immaterial to the system's behavior. At the same time, a higher-level representation, such as the set of statements covered by a set of executions (a common coverage-based metric of test suite quality) is likely to fail to distinguish between two sets of executions that cover different behavior but happen to execute the same statements in alternate orders. In other words, it is difficult to filter trivial differences without also filtering important ones.

**2. Collecting field-use data is difficult.** It is difficult to collect field data, as it requires

a large number of users who utilize the software in a real-world setting. There are costs associated with collecting field data in terms of computational overhead and storage space that may affect the users' experience. These costs must be weighed against the benefits of collection. Furthermore, collecting data from real users poses a privacy challenge.

I address the first challenge by considering behavioral models based on four metrics at different level of details. The models include two coverage models (based on method and statement coverage, respectively), a mutation model (based on the percentage of mutants killed) and a temporal invariant model (based on the number of kTails invariants covered). I compute differences with respect to each of these models between in-house tests and field executions and evaluate each model's effectiveness at representing behavioral differences. Section 4.2.1 describes these models.

I address the second challenge in two ways. First, I turn to the Massive Open Online Course (MOOC) medium. MOOCs have gained popularity in recent years, and this popularity has provided researchers the opportunity to conduct studies with professional developers taking MOOCs. By targeting MOOCs, I was able to access a relatively large user base and evaluate software already used as part of the course, providing us with realistic executions. Second, I turn to the open-source software movement to access real-world systems that use library software and exercise that library software in realistic ways representative of field executions.

## **4.2 Behavioral Models for Difference Measuring**

### 4.2.1 Behavioral models

I use four behavioral models (Section 4.2.1) to measure the differences (Section 4.2.2) between test and field executions. The four models include a set of source code statements covered by executions, a set of methods covered by executions, a set of mutants killed by executions, and a set of temporal invariants over executed methods that hold over the executions.

**Coverage**, a commonly-used measure of test-suite quality (e.g., [34, 40, 41, 42, 43]), measures the fraction of statements, methods, branches, paths, or other code elements touched by a set of executions. For example, statement coverage of a test suite is the fraction of the source code statements in the system under test that are executed by that test suite. Coverage can typically be computed during execution with a relatively low overhead. My coverage models model behavior exercised by an execution as the set of executed statements and the set of executed methods in that execution.

**Mutation** testing [16] can be used to evaluate the quality of existing tests [44] and create new tests [45]. It generates *mutants* by systematically seeding the program with artificial faults by using a set of mutation operators. Each mutant may behave slightly differently from the original program and represents a potential error a developer may have made. If a test that passes on the program fails on a mutant, the test is said to kill the mutant. My mutation model uses the set of killed mutants in an execution to represent the behavior of that execution. For example, mutants killed by field executions but not by tests represent a difference between field and test behavior.

### **Temporal Invariants.**

I mine invariants at the level of methods executed by a trace. For example, if during an execution method `open` executed just before method `close`, then I would mine the invariant `open → close`. I elected to work at the method level because prior work has argued that method call sequences represent the best cost-benefit tradeoff for reproducing field failures [46]. The inferred invariants are represented as a set of sub-sequences of execution traces. The set difference operation is used to compare the invariants inferred for test executions and those inferred for field executions.

### 4.2.2 Behavioral comparison metrics

For each of the behavioral models considered, I measure the similarity ( $\mathbb{S}$ ) and unidirectional difference ( $\mathbb{D}$ ) between field and test executions:

$$\mathbb{S} = \frac{|covered(field) \cap covered(test)|}{|covered(field) \cup covered(test)|} \quad (4.1)$$

$$\mathbb{D}_{tf} = \frac{|covered(test) \setminus covered(field)|}{|covered(test)|} \quad (4.2)$$

$$\mathbb{D}_{ft} = \frac{|covered(field) \setminus covered(test)|}{|covered(field)|} \quad (4.3)$$

where  $covered(field)$  is the set of entities (e.g., statements, methods, mutants, or invariants) covered by field executions and  $covered(test)$  is the set of entities covered by tests.

Similarity ( $\mathbb{S}$ ) measures the fraction of the entities that are common to the field and test executions. By contrast, difference ( $\mathbb{D}$ ) measures the fraction of the entities present in one set of executions that are not present in the other set. For example,  $\mathbb{D}_{ft}$  is the fraction of the behavior exercised in the field that is not exercised by the tests.

### 4.3 Experiment Methodology

My study aims to answer three research questions:

**RQ1:** How does in-field behavior differ from the behavior exercised by developer-written tests?

**RQ2:** Does augmenting developer-written tests with automatically-generated tests help make in-house test suites more representative of field behavior?

**RQ3:** Which of the four behavioral models considered are most effective for comparing developer-written tests and field executions?

For RQ1 and RQ2, I computed each of the four behavioral models considered (see

Table 4.1: Study benchmarks.

<b>benchmark</b>	<b>#methods</b>	<b>LOC</b>	<b>#tests</b>	<b>stmt. cov.</b>
JetUML	603	8,836	*	79.9%
unit tests			53	26.4%
system tests			97	72.9%
Commons IO	940	9,682	1,125	88.8%
Commons Lang	2,647	25,570	3,735	93.3%
Log4j	1,874	21,326	591	76.8%

Section 4.2.1) for each of the four benchmarks described in Section 4.3.1.<sup>1</sup> I then computed the  $\mathbb{S}$  and  $\mathbb{D}$  metrics (see Section 4.2.2). I also manually qualitatively examined the behavior identified by the models as being exercised in the field but not by the tests.

For RQ2, I also augmented the developer-written test suites for my benchmarks with EvoSuite [11], an automated input generation tool for Java programs that is used frequently in software engineering research (e.g., [47, 48, 49]). For each benchmark, I ran EvoSuite five times with different seeds, its default configuration, and a 5-minute time limit per class. This process generated five test suites per benchmark, where each of the test suites took between 7 to 12 hours to generate. I then combined each of the generated test suite with the developer-written tests and obtained five augmented test suites for each benchmark. Finally, I built the four behavioral models for the augmented test suites and compared them with models built for field executions using metrics mentioned in Section 4.2.2.

For RQ3, I compared the behavioral differences between field executions and developer-written tests found by my four models. I measured how much extra behavior exercised in the field each of the four models could find. I also identified which models revealed behavioral differences between the in-house tests and a field execution that triggered a defect in one of the benchmarks (JetUML).

### 4.3.1 Software Benchmarks

Table 4.1 describes the four benchmarks used in my study. JetUML (<http://cs.mcgill.ca/~martin/jetuml>) is a mature editor for UML diagrams. To obtain field uses for JetUML, I had 83 human subjects use JetUML Version 0.7 to design class diagrams for their course projects. The subjects were students enrolled in Georgia Tech’s Online MS in CS (OMSCS) program (<http://www.omscs.gatech.edu>) whose participants are predominantly professional developers. My study did not introduce JetUML to the students; rather, I selected to use JetUML in part because it was already used regularly in the class prior to my study. I believe that uses of JetUML in this setting can be considered realistic field uses, as they involve real users that utilize the tool to create an actual design.

The version of JetUML I used has two test suites (this changed for later versions of JetUML): a JUnit test suite and a set of system tests, both written by JetUML’s developers. The JUnit tests exercise the functionality of JetUML’s underlying framework. The system tests consist of a set of steps to be executed manually. As the instructions describe a single, continuous execution, I consider this as a single system test, made up of 97 steps that exercise the major functionality of the GUI, such as node and edge creation and manipulation. Table 4.1 summarizes the size of JetUML and its two test suites. Together, the two test suites achieve a relatively high coverage, with the system tests achieving a considerably higher coverage than the unit tests, so I consider JetUML well tested in-house.

Apache Commons IO, Apache Commons Lang, and Apache Log4j are software libraries that provide other programs well-defined APIs. I call *client projects* other software that uses these libraries and consider these uses field uses. The three libraries I selected are widely popular and in the top 1% of projects in terms of the number of client projects on Github [50]. Moreover, these libraries have developer-written unit test suites with statement coverage of 77% or above. Table 4.1 provides summary information on the libraries and

---

<sup>1</sup>I did not analyze mutation models for JetUML due to a limitation of the record-and-replay tool I used to log JetUML field executions, which prevented me from replaying executions on mutated code.

their test suites.

**Collecting field executions.** JetUML and the library benchmarks required different methods for collecting field executions. For JetUML, I collected field data by recording the executions performed by real users as they used the software. To do so, I used Chronon (<http://chrononsystems.com>), a record-replay tool for Java programs. During replay, in particular, I used the Post Execution Logging plugin of Chronon to log traces of executed methods and statements. Overall, 147 human subjects submitted their recorded JetUML executions. I filtered out submissions that were missing metadata needed to replay the executions or consisted entirely of opening and immediately closing the application, leaving me with field execution data from 83 human subjects.

To collect field executions for the three library benchmarks, I selected five open-source client projects for each library. To do so, I considered the results of a search on GitHub for projects whose Maven build file (`pom.xml`) listed a dependency on that library. For example, for Log4j, I searched for “pom.xml contains `<artifactId>log4j</artifactId>`”. Maven is one of the most popular Java build tools [51] and all the benchmark libraries I considered use Maven. I thus looked for client projects that build with Maven as well to simplify my data collection process. I eliminated projects that did not have a README file with instructions on how to build and use the software, did not compile with the latest version of the benchmark library, or did not have tests that exercised the library. I kept the first five projects in the order of GitHub’s search results that satisfied these criteria.

I then ran the client projects’ tests. I consider these tests a reasonable proxy for field uses of the libraries, as they use the library as is needed by the client project [5]. Critically, these executions are not library test suites but rather tests of the projects that use the libraries, and thus accurately represent how the libraries are used in the field.

**Building models.** To compute method and statement coverage for my coverage analysis, I used Cobertura (<http://cobertura.github.io/cobertura/>), a widely used code coverage tool. To record method sequences and build invariant models with InvariM-

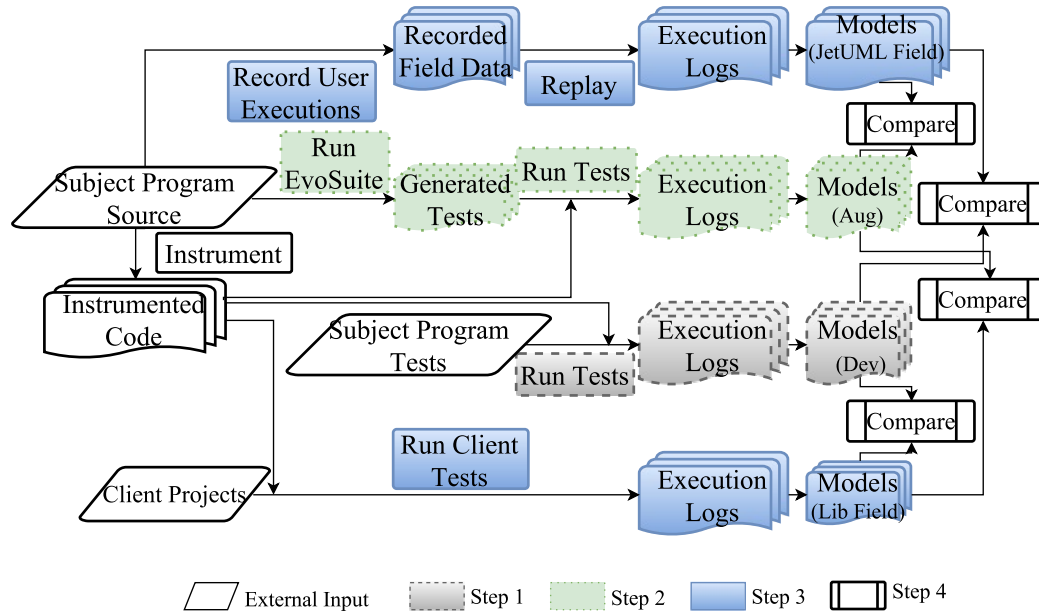


Figure 4.1: Overview of the experiment setup: field data collection, model inference, and execution comparison.

int [2, 3], I built an instrumentation engine on top of ASM (<http://asm.ow2.org/>). To generate mutants, I used the Defects4J framework [52]. Defects4J mutation analysis is built on top of the MAJOR mutation framework [53]. I adopted Defect4J’s default mutation operators, which include operator replacement, statement deletion and literal replacement. To check if client project executions killed mutants, I ran the tests of the client projects using mutated versions of the libraries.

It is worth noting that, because JetUML is a comprehensive UML editor that supports the creation and editing of many kinds of diagrams, its tests cover behavior related to all such diagrams. Since the users were only asked to create class diagrams, in my study I selected the JetUML tests that exercised features related to class diagrams. Similarly, for the library benchmarks, most client projects only used a small part of the libraries considered. In the study, I therefore excluded the classes that none of the client projects executed and the mutants therein.

### 4.3.2 Computing Behavioral Comparisons

Figure 5.2 summarizes the methodology for comparing tests and field executions, which consists of four steps:

**Step 1:** For each benchmark, I ran developer-written tests to generate execution traces and build four behavioral models for those tests: (1) a statement coverage model, (2) a method coverage model, (3) a mutation model (only for library benchmarks), and (4) a method-level invariant model.

**Step 2:** I used EvoSuite to augment three of the benchmarks' developer-written test suites (see Section 4.3).<sup>2</sup> I generated five test suites for each system; each augmented test suite consisted of all the developer-written tests and one of the five EvoSuite-generated suites. I then computed, also for these augmented test suites, the four behavioral models considered. (Recall, however, that I did not compute mutation models for JetUML because the record-replay framework did not allow me to replay field executions on mutants.)

**Step 3:** For each benchmark, I collected field executions and computed my four behavioral models for those executions.

**Step 4:** For each of the four types of models, I computed the similarity ( $\mathbb{S}$ ) and difference ( $\mathbb{D}$ ) metrics (see Section 4.2.2) between field and developer-written test executions, and field and automatically-augmented test executions.

### 4.3.3 Publicly Released Dataset

My dataset includes:

- Source code and developer-written unit and system test cases for JetUML Version 0.7.
- Recorded JetUML field executions from 83 real users. The recorded executions can be used to extract code coverage, execution traces, and program states.<sup>3</sup>

---

<sup>2</sup>The version of Log4j I used is incompatible with the version of EvoSuite that generates Java 7 test suites, which are required by the Major mutation framework.

<sup>3</sup>The executions can be replayed using Chronon Time Travelling Debugger (<http://chrononsystems.com/products/chronon-time-travelling-debugger>).

- Source code and test suites of Commons IO, Commons Lang, Log4j, and the client projects I used in my study.
- Test suites automatically generated by EvoSuite (five per benchmark).
- Instructions on how to instrument, configure, and run the benchmarks.

My dataset, available at [54], can be used for analyzing the difference between test and field execution. I believe that it can also be used for studying other software engineering tasks, such as debugging and test generation.

#### 4.4 Results and Discussion

**RQ1:** How does in-field behavior differ from the behavior exercised by developer-written tests?

Table 4.2 shows the computed  $\mathbb{S}$  and  $\mathbb{D}$  metrics for the four models for each of the four benchmarks.

**Statement coverage.** The similarity between field and developer-written test executions ranges from 18.3% (Commons Lang) to 62.1% (JetUML), with an average of 36.9%. Similarity can be low when one of the sets of executions does not exercise much of the system behavior. Because the field executions that I considered involve a relatively small subset of the user population (both in the case of actual JetUML users and in the case of client projects for the considered libraries), it is not surprising that they use only a subset of the functionality (e.g., methods) of the classes considered. This is also confirmed by the relatively high  $\mathbb{D}_{tf}$  measures, which are between 31.0% (JetUML) and 81.7% (Commons Lang), with an average of 60.7%. Note that higher fractions of statements covered in testing that are not covered in the field result in lower similarity values.

The fraction of statements covered in the field that are not covered in testing, conversely, is relatively low, ranging from 0.3% (Commons Lang) to 13.0% (JetUML), with an average of 6.2%. Thus, overall, the statement coverage metric shows that field executions do differ

from test executions, with 6.2% of the statements executed in the field not having been executed during testing.

For JetUML, system tests are much more similar to field executions than unit tests are (62.2% vs. 8.3%), but still 16% of the statements executed in the field are not executed by system tests. Combining unit tests and system tests reduces the similarity only by 0.1% but reduces the untested statements by 3.0%, supporting the intuition that system and unit tests complement each other. JetUML's unit tests have low coverage (recall Table 4.1) so it is not surprising that they miss statements covered in the field.

To get a better idea of the specific field behavior that in-house test may miss, I looked at the details of JetUML's statement coverage. I observed that often tests missed exceptional or corner cases. For example, they missed exception-handling code, cancellation of operations, and null inputs. Although it is not totally surprising that developers may miss special cases when testing, it is useful to see this confirmed in my results.

**Method coverage.** The fraction of methods exercised by both developer-written tests and field executions ranges from 22.6% (Commons Lang) to 64.9% (Log4J), with an average of 40.9%. The similarity is again low because the field executions do not use many parts of the systems considered. Again, the fraction of methods executed in testing that are not executed in the field is high, averaging 56.1%. The fraction of statements executed in the field that are not executed during in-house testing, conversely, ranges from 0.8% (Commons Lang) to 15.4% (JetUML), with an average of 7.7%, slightly higher than what I observed for statement coverage. Thus the method coverage metric also finds that field executions do differ from test executions.

As with statement coverage, for JetUML, system tests are more similar to field executions than unit tests (57.1% vs. 26.9%), but are better combined. Adding unit tests to the system tests reduces the fraction of methods executed in the field that are not executed by the tests from 21.2% to 15.4%. In addition to the relatively low coverage of the unit tests (recall Table 4.1), the unit tests also call methods directly, bypassing GUI event handling that takes

Table 4.2: Comparison between developer-written tests and field executions.

tests	statement			method			mutation			temp invs		
	S	$D_{tf}$	$D_{ft}$	S	$D_{tf}$	$D_{ft}$	S	$D_{tf}$	$D_{ft}$	S	$D_{tf}$	$D_{ft}$
UML unit	0.08	0.75	<b>0.89</b>	0.27	0.53	<b>0.61</b>				0.05	0.68	<b>0.94</b>
UML sys	0.62	0.29	<b>0.16</b>	0.57	0.33	<b>0.21</b>				0.13	0.52	<b>0.85</b>
UML both	0.62	0.31	<b>0.13</b>	0.52	0.43	<b>0.15</b>				0.15	0.59	<b>0.81</b>
IO	0.22	0.78	<b>0.01</b>	0.25	0.75	<b>0.02</b>	0.47	0.47	<b>0.18</b>	0.11	0.88	<b>0.35</b>
Lang	0.18	0.82	<b>0.00</b>	0.23	0.77	<b>0.01</b>	0.30	0.70	<b>0.01</b>	0.09	0.90	<b>0.24</b>
Log4j	0.46	0.52	<b>0.11</b>	0.65	0.29	<b>0.12</b>	0.15	0.85	<b>0.07</b>	0.14	0.80	<b>0.70</b>
average	0.37	0.61	<b>0.06</b>	0.41	0.56	<b>0.08</b>	0.31	0.67	<b>0.09</b>	0.13	0.76	<b>0.53</b>

place during field executions, and thus fail to exercise GUI interactions.

Also in this case, I looked at the details of JetUML’s method coverage in more detail to get a better understanding of the behavioral differences between in-house tests and field executions. In this case, I noticed that the tests missed many wrapper methods that encapsulate calls to external libraries, which may be due to the fact that developers trusted the libraries and considered the wrapper code to be too simple to test. In addition, in-house tests missed some methods due to different environments in which field executions are performed.

**Mutation.** The fraction of mutants killed by both developer-written tests and field executions ranges from 15.1% (Log4j) to 47.4% (Commons IO), with an average of 30.8%. The similarity results for mutation were higher than those for coverage for some projects, and lower for others. The mutation model captures at least all the statement-coverage behavioral differences, in that mutations in a statement cannot be killed by the tests if the tests do not cover that statement. However, because mutants are not evenly distributed among statements, there is no expectation of a direct relationship between coverage and mutation measures. As before, the field executions do not cover large parts of the systems: 67.4% of the mutants killed by the tests are not killed by the field executions.

The fraction of mutants killed in the field that are not killed by in-house testing ranges from 1.2% (Commons Lang) to 18.2% (Commons IO), with an average of 8.6%, which

is slightly higher than what I observed for the coverage metrics. Therefore, the mutation similarity metric finds that field executions differ from test executions. And if killing mutants is representative of revealing real-world defects [44], 8.6% of the potential defects the field executions encounter would not be caught by in-house testing. This suggests a stronger notion of behavioral differences between field executions and in-house testing than the coverage metrics do.

Mutants killed in the field can survive the tests in two ways: the tests may not execute the mutated line or the tests may execute the line but not trigger anomalous behavior. The first case accounts for 83% of these mutants in my experiments. Both cases can help developers improve test suites; the first reveals an important line to cover with a new test, and the second provides a new input domain uncovered by the existing tests.

**Temporal Invariants.** The coverage and mutation models show average similarity between developer-written tests and field executions between 30.8% and 40.9%. They also show that 6.2% to 8.6% of the behavior exercised in the field was not exercised by the tests. The temporal invariant model shows much starker behavioral differences. The similarity in the temporal invariants mined from the executions ranges from 9.1% (Commons Lang) to 16.8% (JetUML), averaging 12.7%. Of the invariants observed in the tests, from 47.0% (JetUML) to 90.1% (Commons Lang) were not observed in the field. Meanwhile, between 24.3% (Commons Lang) and 80.5% (JetUML) of the invariants observed in the field were not observed during in-house testing. Despite the high coverage achieved by the JetUML test suite, in particular, 80.5% of the in-field behavior, as captured by behavioral invariants, did not occur during testing. Even for Commons IO and Commons Lang, for which test suites failed to cover less than 3% of the statements and methods that executed in the field, 35.4% and 24.3%, respectively, of the invariants observed in the field were not observed during testing. On average, 52.6% of the in-field invariants did not occur during testing, suggesting the test suites are very different from the field executions in terms of the temporal relationships between method executions.

I examined the behavior exercised in the field but not by the tests for JetUML, for which the difference is most pronounced. I found that such behavior occurs for three reasons: (1) field executions exercise code that is not covered by the tests; (2) users perform operations in a different order than the tests; and (3) users perform operations in a different program state than the tests. Most of the cases are of the 2<sup>nd</sup> variety, suggesting that some operation order is interchangeable, and that the developers who write tests sometimes incorrectly assume the order in which users perform operations. For example, all JetUML tests assume that every copy operation is immediately followed by a paste, whereas users sometimes performed a copy, cut, paste instead. This order of operations was never tested in-house. While such ordering may not affect functionality in some cases, when answering RQ3 I will show an example of a bug discovered in the field that the test suite missed for exactly this reason.

**RQ2:** Does augmenting developer-written tests with automatically-generated tests help make in-house test suites more representative of field behavior?

Table 4.3: Augmented tests.

<b>Benchmark</b>	<b>#Generated tests</b>	<b>Augmented coverage</b>
UML	768	87.2%
IO	2,148	92.5%
Lang	4,769	97.1%

For each of the systems considered in my study, I generated five test suites using EvoSuite (recall Section 4.3), augmented the developer-written tests with each of the five generated suites, and again measured the  $\mathbb{S}$  and  $\mathbb{D}$  metrics comparing the field executions with the augmented test suites. Table 4.3 presents information about the augmented test suites for each benchmark, and Table 4.4 shows the comparison results; each cell represents the mean over the five augmented test suites for each benchmark.

The augmented test suites improves the statement coverage of the developer-written test suite for JetUML by 7.3%, for Commons IO by 4.8%, and for Commons Lang by 3.8%.

Table 4.4: Comparison between augmented tests and field executions.

	statement			method			mutation			temp invs		
	S	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	S	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	S	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	S	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$
UML	0.58	0.37	<b>0.11</b>	0.59	0.36	<b>0.12</b>				0.17	0.60	<b>0.77</b>
IO	0.17	0.83	<b>0.01</b>	0.24	0.76	<b>0.02</b>	0.43	0.50	<b>0.18</b>	0.09	0.90	<b>0.33</b>
Lang	0.17	0.83	<b>0.00</b>	0.22	0.78	<b>0.00</b>	0.29	0.81	<b>0.01</b>	0.08	0.92	<b>0.22</b>
average	0.17	0.83	<b>0.00</b>	0.23	0.77	<b>0.01</b>	0.36	0.66	<b>0.10</b>	0.09	0.91	<b>0.28</b>
<b>average (dev)</b>	0.20	0.80	<b>0.00</b>	0.24	0.76	<b>0.02</b>	0.39	0.59	<b>0.10</b>	0.10	0.89	<b>0.30</b>

For JetUML, the augmented test suites have a slightly lower similarity with field executions with respect to the statement coverage (58.3% vs. 62.1%). The decrease is due to the generated tests covering more code that the field executions do not exercise. With respect to method coverage and invariant models, the similarity is higher (59.3% vs. 51.6% and 17.4% vs. 15.3%, respectively). The increase is caused by the fact that EvoSuite-generated tests are able to cover some field-executed methods that are missed by developer-written tests.

For Commons IO and Commons Lang, the augmented tests result in decreased similarity between test runs and field executions with for all four models (see the last two rows of Table 4.4). Also in this case, the change is due to the generated tests covering additional code, mutants, and invariants that field executions did not exercise.

The augmented test suites capture more in-field behavior than the developer-written test suites for statement coverage, method coverage, and invariant models for all benchmarks. With respect to the statement coverage model, the augmented tests miss 10.9% and 0.398% of the in-field behavior (for JetUML and libraries, respectively), as compared to 13.0% and 0.403% for the developer-written tests. For the method coverage model, the augmented tests miss 3.9% and 0.8% of the in-field behavior, as compared to 15.4% and 1.5% for the developer-written tests. For the mutation model, conversely, there is no change in terms of missed mutants covered in the field executions (both 9.7%). Finally, for the invariant model, the augmented tests miss 76.7% and 27.7% of the in-field behavior, as compared to 80.5% and 29.9% for the developer-written tests. Overall, while the representativeness of the tests

is improved, there are still significant differences with the in-field behavior.

It is worth mentioning that the EvoSuite-generated tests cover more of the exceptional behavior than the developer-written tests. This is the main reason why the augmented test suites reduced the number of statements covered in the field but missed by the tests. However, the new tests that cover exceptional flows are mostly trivial and typically consisted of passing null values; most other control flows related to corner cases are not covered by the new tests. Also, EvoSuite-generated tests were unable to kill mutants that are missed by developer-written tests. With respect to invariants, new invariants induced by the new tests are largely due to the fact that EvoSuite checks a value after setting it whenever possible. This is the case, for instance, for the new invariant `GraphPanel.setModified → GraphPanel.isModified`. These invariants are likely not as useful for capturing important field behavior as those mined from developer-written tests.

Overall, coverage driven automated test generation helped only marginally in reducing the differences between field executions and in-house tests. Most importantly, it mainly helped in the trivial case of code for which no developer tests existed.

**RQ3:** Which of the four behavioral models considered are most effective for comparing developer-written tests and field executions?

Overall, all models were able to detect some differences between test and field executions. The mutation models identified more differences than the statement and method coverage models (Tables 4.2 and 4.4). In general, there is not a subsumption relationship between mutation and structural coverage. On the one hand, statement and method coverage models can reveal differences in behavior that mutation does not detect; for an example, consider the case of a statement that is covered in the field but not in house and is not selected for mutation. On the other hand, mutation models can reveal differences in behavior that statement and method coverage would miss; consider, for instance, the case of a statement that is mutated, is covered both in the field and in house, and only in the field is executed under a state that kills the mutant.

The invariant model is strictly more inclusive than method coverage models: if an invariant involving a method is mined, that method must have been executed. Using invariant models to characterize differences between field and test executions cannot therefore miss any information reported by method coverage models. Statement coverage models could identify finer-grained differences than method-level invariants report, but the data in Table 4.2 suggest that this was uncommon. (I did not consider statement-level temporal invariant models, which would capture all such differences, but would also be impractically large.) In general, the invariant model finds more sophisticated differences than the coverage models, such as different orders of method executions, and the execution of the same methods under different program states.

I found some evidence that the differences between field and test executions that the invariants model finds but other models miss can be important. For a concrete example, when starting JetUML, users have to either select a type of diagram to create, or use the `File` menu to open or create a file. The invariant model found that users sometimes selected `Undo` from the `Edit` menu as the first action. This unexpected operation caused an exception that crashed the program. Neither developer-written nor EvoSuite-augmented test suites found this error, and the coverage and mutation models did not identify this difference. The JetUML developers have identified this as a real defect and have fixed it. In more general terms, my results suggest that (1) invariant models may be better than simpler models at discovering important behavioral differences between in-house and field executions and (2) these differences can reveal relevant behavior (including defects).

Here is a summary of my findings for the systems and executions I considered:

- Field executions can differ considerably, in terms of the behavior they exercise, from in-house test executions.
- Automatically generated tests can only marginally improve the representativeness of in-house tests.
- All behavioral models can find differences between in-house testing and field execu-

tions. Specifically, statement coverage models can identify corner cases missed during testing, while method coverage models can find high-level differences in the features used. Mutation models can miss some differences identified by the simpler coverage models, but they may be able to identify specific states not covered by the tests. Finally, invariant models subsume (at the method level) coverage models and can identify richer differences, such as differences in operation order and context.

- Unsurprisingly, using the state of the practice (coverage) or the state of the art (mutation) to assess the quality of a test suite falls short of precisely measuring how well the test suite represents field executions. An invariant-based model may be a better adequacy and selection criterion, but further investigation on infeasibility issues is required.

## CHAPTER 5

### MIMICKING USER BEHAVIOR USING FIELD EXECUTION DATA

#### 5.1 Motivating Example

Figure 5.1 shows an example of behavior exercised in the field but not tested in-house and illustrates why it is non-trivial to mimic it. The example is a simplified snippet of one subject program I use in the evaluation.

The main purpose of the program is to parse a file with a specific format and it consists of two functions: *main* and *parse\_expr*. It takes an input file from the command line and parses the content of the file to define properties that it uses later. It reads file content after “[” into a char array *tmp\_str* (line 36) until it encounters “]”, “#” or “\n”. It then checks the value of *tmp\_str* and handles the string accordingly (lines 39-49). If the value starts with the word “property”, then a property is defined (line 39-41) and its value is set to the content after “property”. If *tmp\_str* starts with “if”, one or more properties are used as a condition and the function *parse\_expr* is called to analyze the condition (lines 43-46). *parse\_expr* first checks whether the condition is compound by examining the existence of the *or* operator (“ || ”) (lines 7-21). If it is a compound condition, variable *prop* takes the value of the first sub-condition; otherwise it takes the value of the entire expression. Afterwards the function checks whether the property used (*prop*) is already defined or not (lines 23-26). It returns the defined property or prints out an error message.

Consider a file with the following content:

```
[property a]
[property b]
[property c]
[if a || (b || c)]
```

```

1 #include <stdio.h>
2 #include <string.h>
3 unsigned prop_num = 0;
4 char *props[64];
5 char *parse_expr( char * expr) {
6     char *prop, *position, *open, *close;
7     if ((position = strstr( expr, " || ")) != NULL ) {
8         open = strchr( expr, '(');
9         if (open != NULL ) {
10             close = strchr( expr, ')');
11             size_t len = close - open - 1;
12             strncpy(prop, open + 1, len);
13             prop[len] = '\0';
14         } else {
15             size_t len = position - expr;
16             strncpy(prop, expr, len);
17             prop[len] = '\0';
18         }
19     } else {
20         prop = expr;
21     }
22     for (int i = 0; i < prop_num; i++) {
23         if ( strcmp(prop, props[i]) == 0) {
24             return props[i];
25         }
26     }
27     printf("Property %s is not defined.\n", prop);
28     exit(1);
29 }
30
31 int main (int argc, char *argv) {
32     FILE *f = fopen (argv[1], "r");
33     char buffer[256];
34     char *tmp_str = &buffer[0];
35     if ( f != NULL) {
36         while ( fscanf(f, " [ %[^#\n] ]", tmp_str) > 0) {
37             char prop[64];
38             sscanf(tmp_str, "%s", prop);
39             if ( strcmp(prop, "property") == 0) {
40                 tmp_str = tmp_str + 8;
41                 props[prop_num++] = malloc (strlen(tmp_str));
42             }
43             if ( strcmp(prop, "if") == 0) {
44                 if ( sscanf( tmp_str, "%*s %[^]", prop ) > 0 )
45                 {
46                     parse_expr(prop);
47                 }
48             }
49         }
50         fclose(f);
51     } else {
52     }
53 }

```

Figure 5.1: A motivating example.

Executing the program with the file covers an invariant sequence `12:strncpy` → `23:strcmp` → `36:fscanf`, where `12:strncpy` means a method call to *strncpy* in Line 12. This behavior is not tested by the in-house test suite.

To trigger this behavior, the content of the input file needs to meet the following requirements: (1) It should start with “[” and end with “]”; (2) the content between “[” and “]” (*tmp\_str*) should not contain “]”, “#” or “\n”; (3) *tmp\_str* should have at least two parts separated by white space; (4) the first part should be “if” and the second part should contain “||” and parentheses; and (5) the property between the parentheses should already be defined. The example program itself has several different branches in a while loop; the library functions called in the program, *fscanf* and *sscanf* in particular, consist of layers of nested loops and branches. Traditional symbolic-execution-based test generation tools need to explore an exponential number of paths before finding an input that can cover this behavior. However, if we give the symbolic execution engine some program points as targets to reach, these targets will guide the engine towards a path that exercises the intended behavior and reduce the search space significantly.

## 5.2 An Approach for Mimicking Executions

### 5.2.1 Overview

As mentioned in Chapter 1, our goal is to bridge the gap between in-house tests and field executions by discovering untested field behavior and generating new inputs that exercise such behavior. To achieve this goal, I propose Replica, a technique that collects execution data from the field, identifies untested behavior by comparing field data with in-house test-execution data, and generates executions that exercise such behavior. Figure 5.2 shows the high-level overview of the technique. Inputs and outputs to Replica are elements in a parallelogram with a solid border and in a rounded rectangle, respectively. Elements in a parallelogram with a dashed border are intermediate outputs and those in a rectangle are Replica’s components.

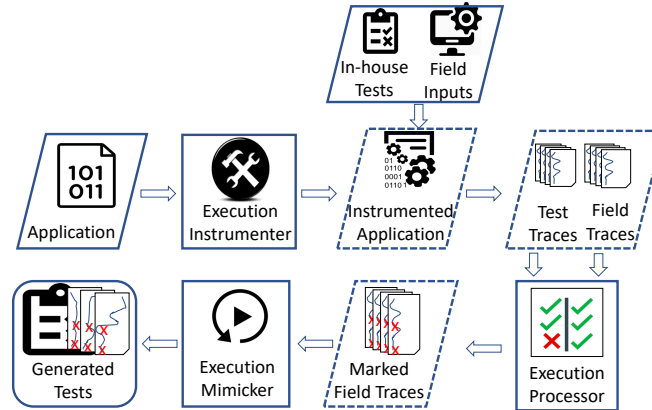


Figure 5.2: High-level view of Replica.

Replica has a main component, *Execution Mimicker*, and two assisting components, *Execution Instrumenter* and *Execution Processor*. The Execution Instrumenter takes the source code of the subject program and produces an instrumented executable. The executable is tested in-house and deployed to the field. Then the execution traces from both test and field executions are collected. Execution Processor compares the collected traces and marks differences in the traces. Finally, Execution Mimicker uses the marked traces as guidance to generate inputs that mimic field executions. I will discuss each component in this section.

### 5.2.2 Execution Instrumenter and Processor

Various execution information can be collected from the field and used for generating test cases. As shown in Chapter 4, field executions differ from developer-written tests most significantly in terms of invariant violations. More specifically, the study mined a temporal invariant, called kTails, on method call sequences. I take advantage of the findings from the study and use method call sequences as execution data and mine kTails invariant from the collected data.

To collect method call sequences for a program, Execution Instrumenter inserts probes at each call site of P before deploying it. These probes record all method calls that happen during the execution and generate a trace consisting of a call sequence, which is dumped at the end of the execution. In addition, Execution Instrumenter also adds a data transfer

function at each program exit point to send the generated trace to a server (when the instrumented program is running in the field).

Execution Processor takes as input the traces generated by the instrumented program and mines kTails invariants from those traces. The invariants computed from in-house test-execution traces are stored for future comparison with field invariants and those mined from field executions are compared with previously stored test invariants. Invariants (i.e., behaviors) exercised in the field but not in house are suitably marked.

### 5.2.3 Execution Mimicker

The core component of Replica is Execution Mimicker, whose essential approach is to incrementally guide symbolic execution. Our approach differs from traditional guided symbolic execution in three aspects. First, our approach does not aim to reproduce the exact trace guiding the input generation; instead, it gives executions the freedom to explore different paths as long as they cover the important subsequences. Second, in terms of searching strategy, it uses the guidance incrementally by attempting to reach the important targets first and then adding intermediate points on the fly when the attempt fails. Finally, the results of our algorithm are not necessarily a single input that covers all intended targets; the results can be a set of test inputs, each covering one or more subsequences in the execution trace that are not exercised by in-house tests.

The key inputs to Execution Mimicker are a subject program  $P$ , represented as an interprocedural control flow graph (*icfg*) and a marked execution trace  $mTrace$  collected by Execution Instrumenter when running  $P$  in the field.  $mTrace$  is analyzed by Execution Processor so that its entries corresponding to invariants exercised only in the field are marked. Each entry in  $mTrace$  is a pair  $\langle t, v \rangle$  where  $t$  is a potential target method call and  $v = 1$  if  $t$  is in an invariant-violating subsequence,  $v = 0$  otherwise. In addition, the algorithm takes in a timeout threshold *timeout* indicating that, if exceeded, the component should add an intermediate target (explained below) to reach. The output of Execution Mimicker is a set

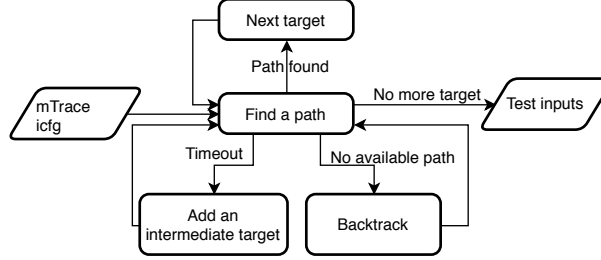


Figure 5.3: Execution Mimicker.

of test inputs that exercise subsequences marked in  $mTrace$ .

Figure 5.3 demonstrates on a high level how the incremental, guided input generation works. Execution Mimicker first tries to find a path that reaches entries in  $mTrace$  that are marked as invariant-violating. If it cannot reach a targeted point within the time limit (given by a customizable parameter  $timeout$ ), it adds an entry in  $mTrace$  that is not invariant-violating as an intermediate point (see Algorithm 3) and first finds a path from the target that is last reached to the added point then resumes finding a path from the added point to the target point. If no path is available to reach the current target, Execution Mimicker backtracks to a previous target to explore new states. The process iterates until the last invariant-violating point is reached or all possible program states have been explored. Algorithm 1 describes the process in detail.

$mapInst$  (line 1) finds the mapping between the method calls in  $mTrace$  and the instructions in  $icfg$  and marks the invariant-violation instructions. Note that for an instruction marked as invariant-violating, I also mark its dominators recursively. After the mapping, each method call is represented by an instruction; I use the term “instruction” to refer to a method call in the following description. The function also finds out the first and last marked target ( $firstTarget$  and  $finalTarget$ , respectively) in the trace so that Execution Mimicker knows what is the first target to reach and when to terminate the search.  $init$  (line 2) initializes the search space  $states$  by adding the initial program state at the entry point. Each state consists of the last instruction ( $state.inst$ ) it reached so far and a path condition  $state.const$  it accumulated to reach the instruction. It initializes the current target  $curTarget$  to  $firstTarget$ . The flag  $isImmediate$  indicates whether the current target

```

Input :  $mTrace$  : An ordered pair list  $\langle t_i, v_i \rangle$ 
          $icfg$  : ICFG for program P
          $timeout$ : Timeout to reach a target
Output:  $testInputs$  : A set of test inputs
1:  $mapInst(icfg, mTrace, firstTarget, finalTarget)$ 
2:  $init(states, curTarget, isImmediate, start, max)$ 
3: while  $states \neq \emptyset$  do
4:    $state \leftarrow SelState(icfg, states, curTarget)$ 
5:   if  $state == null$  then
6:     if  $curTarget == markTrace[0].first$  then
7:        $mTrace[max].second \leftarrow 0$ 
8:       if  $max == mTrace.index(finalTarget)$  then
9:         return  $testInputs$ 
10:      end if
11:       $curTarget \leftarrow$  next marked target
12:      if  $max < mTrace.index(curTarget)$  then
13:         $max \leftarrow mTrace.index(curTarget)$ 
14:         $isImmediate \leftarrow false$ 
15:      end if
16:    else
17:       $curTarget \leftarrow$  previous marked target
18:    end if
19:  end if
20:  while  $state \neq null$  and  $state.inst \neq curTarget$  do
21:    if  $state.inst$  is a call instruction and  $isImmediate$  then
22:      continue
23:    end if
24:     $executeInst(state.inst)$ 
25:    if none of the successors of  $state.inst$  are reachable then
26:       $sol \leftarrow solver.getSol(state.const)$ 
27:       $testInputs.add(sol)$ 
28:       $states.remove(state)$  break
29:    end if
30:    if  $curr\_time() - start > timeout$  then
31:       $next \leftarrow AddTarget(mTrace, reached, curTarget)$ 
32:       $start \leftarrow curr\_time()$  break
33:    end if
34:  end while
35:  if  $curTarget == finalTarget$  then
36:     $sol \leftarrow solver.getSol(state.const)$ 
37:    if  $sol \neq null$  then
38:       $testInputs.add(sol)$ 
39:      return  $testInputs$ 
40:    else
41:       $states.remove(state)$  continue
42:    end if
43:  else
44:     $reached \leftarrow curTarget$ 
45:     $next \leftarrow$  next marked target
46:    if  $next$  immediately follows  $curTarget$  then
47:       $isImmediate \leftarrow true$ 
48:    else
49:       $isImmediate \leftarrow false$ 
50:    end if
51:     $curTarget \leftarrow next$ 
52:    if  $max < mTrace.index(curTarget)$  then
53:       $max \leftarrow mTrace.index(curTarget)$ 
54:       $start \leftarrow curr\_time()$ 
55:    end if
56:  end if
57: end while
58: return  $testInputs$ 

```

**Algorithm 1:** GenerateInput

should immediately follow the previous one (if they are consecutive in  $mTrace$ ) and it is initialized to false. The initialization also starts a timer using  $start.max$  is used keep track of the furthest target in  $mTrace$  the algorithm tries to reach and it is initialized to the index of  $curTarget$ .

The iteration starts from selecting a state to explore (line 4) by calling  $SelState$  defined in Algorithm 2.  $SelState$  selects a state whose instruction has the shortest distance to the current target to reach. It iterates over all candidate states and computes the shortest distance between their instructions and  $curTarget$  in terms of the number of statements in  $icfg$  (lines 3–9) in Algorithm 2. If none of the explored states can reach  $curTarget$ , the algorithm returns  $null$ .

**Input** :  $icfg$  : ICFG for Program P  
            $states$  : A set of symbolic states  
            $curTarget$  : The target to reach  
**Output** :  $selectedState$  : The selected state to explore

```

1:  $minDis \leftarrow \infty$ 
2:  $selectedState \leftarrow null$ 
3: for  $state \in states$  do
4:    $currDis \leftarrow short\_dis(state.inst, curTarget)$ 
5:   if  $currDis < minDis$  then
6:      $minDis \leftarrow currDis$ 
7:      $selectedState \leftarrow state$ 
8:   end if
9: end for
10: return  $selectedState$ 

```

**Algorithm 2:** SelState

If there is no candidate state selected, it means the current target is not reachable from all explored states so far and backtracking is needed to explore new paths (lines 5–19).  $GenerateInput$  sets  $curTarget$  to the previous one and tries to find a different path to reach it and generates new program states. If the algorithm backtracks to the first entry of  $mTrace$  but still fails to find a path, it means the target we want to reach (whose index in  $mTrace$  is  $max$ ) is not reachable from the program entry. Therefore we give up reaching this target and move on to the next target instead. If the target is already the last target in  $mTrace$ , the

algorithm returns the solutions accumulated so far. Note that the *GenerateInput* algorithm does not enforce the reproduction of all subsequences in one execution. Instead it has the freedom to explore other subsequences even when one target cannot be reached.

If a state is returned by *SelState*, the algorithm keeps executing the program along the selected states until it reaches *curTarget* (lines 20–34). If the execution reaches a call instruction that is not *curTarget* (lines 21–23), we check the *isImmediate* flag. If it is true, it means we want to cover behavior  $t_i \rightarrow t_{i+1}$ , but it is not reachable from the currently selected state since it reached another call instructions before reaching  $t_{i+1}$  and thus breaks the invariant, we therefore skip the state and search for another state. Otherwise the execution continues by executing the current reached instruction. If the current state’s instruction is a conditional statement that uses symbolic values, the algorithm checks the satisfiability of each branch’s path condition and forks the current state if necessary. If the instruction is not a conditional statement, the execution performs suitable operations and update the current state’s instruction to its successor. The detail of *executeInst* follows the traditional symbolic execution process and is thus elided from Algorithm 1. If the execution cannot proceed due to unsatisfiable constraints (lines 25–29), the algorithm consults the solver for a solution that covers the furthest reachable path for this state and stores the solution as a candidate input.

While searching for a path to *curTarget*, the algorithm also keeps track of the time spent reaching the current target so far. If *curTarget* cannot be reached within *timeout* (due to the path explosion problem), the algorithm *AddTarget* (defined in Algorithm 3) is invoked to reduce the search space by adding an intermediate point.

*AddTarget* takes in the marked execution trace *mTrace*, the latest target that is already reached *lastReached* and the current target we want to reach *target*. The algorithm computes the geometric mean of *lastReached*’s and *target*’s respective indices and assigns the value to *next* (line 2). If no previous target has been reached yet, or the previous target has index 0, *next* takes the square root of the index of *target*(line 4). Then it retrieves the

<p><b>Input</b> : <math>mTrace</math> : An ordered pair list <math>\langle t_i, v_i \rangle</math>  <math>lastReached</math> : The last target reached  <math>target</math> : The original target to reach</p> <p><b>Output</b> : <math>nextTarget</math> : The next target to reach</p> <p>1: <b>if</b> <math>lastReached \neq null</math> <b>and</b> <math>index(lastReached, mTrace) \neq 0</math> <b>then</b>  2:   <math>next \leftarrow sqrt(index(lastReached, mTrace) * index(target, mTrace))</math>  3: <b>else</b>  4:   <math>next \leftarrow sqrt(index(target, mTrace))</math>  5: <b>end if</b>  6: <math>nextTarget \leftarrow mTrace[next].first</math>  7: <b>return</b> <math>nextTarget</math></p>
--

**Algorithm 3:** AddTarget

instruction whose index is  $next$  in  $mTrace$  and use the instruction as the intermediate target to add.

Note that any point in between  $lastReached$  and  $target$ , such as the middle point computed using an arithmetic mean can be added as the intermediate point and the experiment shows that using the geometric mean works better than using arithmetic mean. The reason is that in most of the field traces, the distance between two targets are relatively long and even the middle point may not be close enough to the last reached target for the searcher to find a path within the given time. By using geometric mean, the intermediate point is closer to the previously reached target and it is easier for the searcher to find a path in between and avoid the necessity to wait for another timeout and add another intermediate target.

Finally, when  $curTarget$  is reached, the algorithm first checks whether  $curTarget$  is the last target ( $finalTarget$ ) in the marked trace. If so (lines 35–42), it passes the constraints to the solver and asks for a solution. If the solver returns a solution, the algorithm adds the solution to the set of generated inputs and return the set as we have covered all the marked subsequences at this point. Otherwise *GenerateInput* discards this state and continues the main loop to explore more available states. If the current target is not the final one (lines 43–55), the algorithm finds the next marked target to reach ( $next$ ). To find the next marked target, the algorithm iterates over  $mTrace$  from the current instruction until it finds the next instruction that is marked. If  $next$  immediately follows  $target$  in the trace, it means

the invariant  $target \rightarrow next$  is exercised in the field but not by the in-house tests and we want the generated test inputs to cover this invariant. We ensure this by setting the flag *isImmediate* to *true*, meaning no other call instruction should be reached before reaching the target. Otherwise *isImmediate* is set to *false*. After this we update our current target to *next*. If the new target is one that the algorithm never attempted to reach, we update the furthest attempted target tracker *max* and reset the timer. The purpose of the *max* tracker is to distinguish whether the algorithm attempts to reach a target for the first time or through backtracking.

The algorithm terminates when there is no additional state to explore (line 58), or there is no additional target to reach (line 9 or line 39). In both cases, the algorithm returns a set of inputs that mimic untested field behavior by exercising partial or all invariant-violating subsequences.

#### 5.2.4 Applying Replica to the Motivating Example

The untested behavior mentioned in Section 5.1 comes from the following trace (represented with line numbers): [32, 36, ..., 38, 39, 43, 44, 46, 7, 8, 10 **12, 23, 36**, ...]. After I compare this trace with the invariants generated from in-house test cases, I identified that [12, 23, 36] is a sequence that never occurred in test executions and thus I mark this sequence in the original trace as invariant-violating. Now the goal is to mimic an execution that covers these three lines consecutively. Note that there are actually thousands of method calls before the sequence [12, 22, 36] and trying to reproduce the entire trace is not an ideal approach. First, constraints collected along the entire trace will get too complicated for the solver to solve. Second, even if the solver is able to solve the constraints, it will take a long time to cover the intended sequence. Therefore, I focus on reaching the invariant-violating entry directly.

To start, Replica performs dominator analysis and marks lines 32, 36, 38, 39, 40, 43, 44, 46, 7, 8 and 10 as targets to be reached because they dominate the invariant-violating line 12. We ask the symbolic execution engine to find a path to reach these marked lines in order

and record the constraints needed along the path. With the given targets, a path can be easily found to cover 12 and 23 because following the marked trace can exactly lead the execution towards lines 12 and 23.

However, it is not trivial to reach line 36 after line 23 because the program should not exit at line 28. To make this happen, line 41 needs to be executed so that a property can be defined. The starting trace does not mark line 41 and Replica may not select a path covering line 41 as the distance between lines 41 and 12 is not the shortest. If within the given time, the intended sequence cannot be reached, we find a target that is not originally marked and add it as an intermediate target before reaching the invariant-violating sequence. In the case of the example, line 41 may not be added as the first intermediate target, but it will be added eventually as we repeatedly add the unmarked points.

### 5.3 Evaluation

In my empirical evaluation, I addressed the following research questions:

- **RQ1: Can replicas exercise the new behaviors missed by in-house tests and observed in the field?** This research question evaluates the *effectiveness* of my incremental, guided technique in mimicking field executions, i.e., in generating tests that violates the same invariants as field executions.
- **RQ2: Are replicas effective in detecting faults that are detected by their originals?** This research question aims to evaluate the *usefulness* of the generated test inputs in terms of their fault detection capability. More specifically, I evaluate whether replicas can kill the same mutants that their original field executions kill.

In addition to the two research questions, I also conducted an experiment to assess the runtime overhead imposed by instrumenting subject programs.

### 5.3.1 Replica Implementation

Replica’s Execution Instrumenter uses the LLVM compiler infrastructure [55] for data collection and cURL [56] for data transfer.

The Execution Processor uses InvariMint [2] to mine invariants from both test and field execution traces and identify invariants that occur in the field but not in the test which are considered as violations.

Replica’s Execution Mimicker is built on top of Klee [10], a symbolic execution engine for C programs. I implemented my own customized searcher based on my Algorithms 1– 3.

### 5.3.2 Program Considered and Field Data Collection

To answer the above questions, I applied Replica on a set of real-world applications.

I considered four subject programs, *TSL generator*, *grep*, *sed* and *gzip*. I chose these programs because they are open source programs that come with developer-written in-house test suites. I collected field executions from the four programs in two ways.

*TSL generator* is used by hundreds of students, most of whom have professional development experience, from a large-scale online class. I instrumented the program with Replica before distributing it to the students. If a student decides not to submit his/her execution information, he/she can turn off data collection by specifying the “–disable-data-collection” option. Otherwise, execution traces are collected and transferred to a server. When a student runs *TSL generator* for the first time, a unique id will be assigned for the machine the application is running on. Different executions from the same machine are distinguished by the timestamps. *Grep*, *sed* and *gzip* are command line tools widely used by real users for a long period of time. I looked into their bug report archives, extracted bug reports that contained explicit test inputs. I ran the subject programs on these inputs as proxy of field executions.

After collecting call sequences from both field and test executions, I used our *Execution Processor* for mining invariants from these traces and comparing the invariants with those

mined from in-house test executions. I used  $k = 2$  for kTails invariants in my experiment.

I then used Milu [57], an open-source C mutation testing tool to generate mutants for my subject program. I applied the default mutation operators: arithmetic assignment, arithmetic operator and relational operator mutation and generated mutants for each subject program<sup>1</sup>. I ran both the in-house tests and field executions against the mutated programs. I used the outputs from running the original program as oracles. If the output from running a mutant is different from the oracle, I say the mutant is killed. Note that I did not perform equivalent mutants detection as it is an expensive process in general and is out of the scope of this dissertation.

Table 5.1 summarizes the information of my subject programs. Column *Size* shows the size of each program in thousand lines of code (kLOC). Column *Test* refers to the number of in-house test scripts for grep, sed and gzip; each test script can contain multiple test executions. Column *Untested Inv* refers to the number (and percentage) of kTails invariants that are missed by in-house tests but covered by field executions. Similarly, column *Missed Mut* means the number (and percentage) of mutants killed by field executions but not killed by in-house tests.

Table 5.1: Subject programs used in the study.

<b>Program</b>	<b>Size</b>	<b>Test</b>	<b>Field</b>	<b>Untested Inv</b>	<b>Missed Mut</b>
<b>tsl</b>	0.7	65	34,958	464 (51%)	107 (28%)
<b>grep</b>	10	110	98	374 (19%)	0 (0%)
<b>sed</b>	14	60	108	2532 (76%)	319 (41%)
<b>gzip</b>	5	5	46	417 (49%)	90 (11%)

The size of our subject programs ranges from about 0.7 kLOC to 14 kLOC. TSL generator comes with 65 in-house test inputs (and their expected outputs as oracles). Grep, sed and gzip contain 110, 60 and 5 test scripts respectively. For TSL generator, I ran the data collection process for two semesters and collected 34,958 individual execution traces from 747 machines. For the other three programs, I examined the bug reports from August

<sup>1</sup>I did not mutate all functions in grep, sed and gzip due to the limitation of Milu.

2013 to December 2018 and extracted 98, 108 and 46 concrete inputs for grep, sed and gzip, respectively. I ran the programs on their corresponding inputs as a proxy of field executions and collected call sequences.

Field executions cover 464 (51%) extra invariants not covered by test executions for TSL generator, 19% for grep, 76% for sed and 49% for gzip. In terms of mutants, 28% of mutants that are killed by field executions survived in-house tests for TSL generator, 41% for sed and 11% for gzip. For grep, all mutants killed by field executions are also killed by in-house tests. This result is consistent with the conclusion from the study in [58], which showed that in-house tests can miss a considerable amount of field behavior.

To reduce the number of traces needed to mimic for TSL generator (recall that there are about 35k traces), I ordered them by the number of violations they exercise and then iterated over the ordered traces. For each trace, if it violates at least one extra invariant, then I add it to the reproduce list; otherwise I exclude it. I have 144 traces after the selection, whose lengths range from 11 method calls to 1.27 million method calls. These 144 traces represent the originals to be mimicked.

### 5.3.3 Experiment Protocol

With the collected and pre-processed data, I set up our experiment to answer our research questions as follows.

#### *Test Generation*

I first ran Klee with the default search heuristics for 48 hours to generate test inputs without any guidance. I then provided collected field traces, with invariant-violating points marked, to Replica as guidance and ran Replica for 1 hour per trace. After I generated new test inputs, I checked the representativeness improvement of the augmented tests in terms of extra invariants exercised and evaluated the quality of the generated tests in terms of the extra mutants they killed.

### *Invariant Analysis*

I executed our subject program with the two sets of generated test inputs for each program and mined invariants from the execution traces. I then compared the invariants with the extra ones exercised in the field and calculated how much extra is covered for each set of generated inputs.

### *Mutation Analysis*

I then ran the two sets of inputs generated in Section 5.3.3 against the mutants. I calculated the number of extra mutants, i.e., mutants that survived in-house tests but killed by field executions, killed by generated inputs.

### *Runtime Overhead Analysis*

I ran each subject program with their collected field inputs twice, once with the uninstrumented version and once with the instrumented version. Note that I did not run all field executions for TSL generator (recall that I have about 35k field executions); instead I ran the program on 100 randomly selected field inputs. I then compared the time used to compute the runtime overhead.

## 5.3.4 Results and Discussion

Table 5.2: Experiment results.

<b>Program</b>		<b>tsl</b>	<b>grep</b>	<b>sed</b>	<b>gzip</b>	<b>Average</b>
<b>Overhead</b>		2%	16%	13%	22%	14%
<b>Untested Behaviors</b>	<b>Baseline</b>	7.8%	1.9%	1.1%	7.9%	4.7%
	<b>Replica</b>	73.7%	50.4%	45.5%	54.4%	56.0%
	<b>Improvement</b>	9.45x	26.5x	41.3x	6.89x	11.9x
<b>Faults Revealed</b>	<b>Baseline</b>	37.5%	-	10.0%	11.1%	19.5%
	<b>Replica</b>	79.4%	-	41.4%	63.3%	61.4%
	<b>Improvement</b>	2.12x	-	4.14x	5.70x	3.15x

Table 5.2 summarizes the results for our experiment. Column *Overhead* shows the extra time needed to run the instrumented version of each subject program compared to running the original program. Column *Untested Behavior* refers to the percentage of untested field invariants that are exercised by inputs generated with *Klee* and *Replica*. Similarly, *Faults Revealed* refers to the percentage of mutants missed by in-house tests that are killed by *Klee*- and *Replica*- generated test inputs.

#### *Runtime Overhead.*

The runtime overhead imposed by instrumenting the program ranges from 2% (TSL generator) to 23% (gzip), with an average of 13%. Given that our current instrumentation uses a naive method that simply logs events whenever they take place, I view these numbers as acceptable. Efficient trace collection techniques such as those based on LL(1) grammar [59] or using hardware support [60] could be used to reduce runtime overhead significantly.

#### *Effectiveness of Replica.*

Test inputs for generated by using Klee’s random search strategy cover 7.8%, 1.9%, 1.1% and 7.9% of all invariants that are missed by in-house tests for each subject program, with an average of 4.7%. After augmenting in-house with tests generated by Klee, there is still a large portion of invariants (>90%) that are untested. The low coverage of the missed invariants is caused by the explosion of feasible execution paths, leaving Klee an extremely large search space to explore. Without proper guidance, Klee keeps exploring paths covering already-tested call sequences.

Replicas, on the other hand, cover 73.7%, 50.4%, 45.4% and 54.4% of untested invariants for TSL generator, grep, sed and gzip respectively, with an average of 56.0%. Although replicas cannot exercise all invariants that are only covered by field executions, they exercise significantly more such invariants than inputs generated by Klee, with an average improvement of 11.9 times.

I manually inspected the invariants that Replica is unable to generate inputs to exercise and found that covering such invariants involves very complicated constraints and the underlying solver fails to find a solution within the given time. For instance, Replica failed to generate inputs that consist of nested compound conditions (e.g., “[if (A && (B || C)) && ((D || E) && F)]”). The highly-structured input itself produces complex constraints. In addition, several sub-strings of the input refer to properties that need to be compared with all previously-defined properties. that are previously-defined. This aggregates several previously independent constraint sets and produces a massive constraint that the solver fails to find a solution for.

In summary, Replica can generate test inputs that exercise a large percentage of extra invariants missed by developer-written tests and it significantly outperforms the techniques using random search and minimal guidance. However, the effectiveness of Replica is constrained by the limitation of the underlying solver.

#### *Usefulness of Replicas.*

As shown in Column *Faults Revealed* of Table 5.2, the percentages of extra mutants (mutants that are killed by field executions but not by in-house test suites) killed by Klee-generated inputs are, on average, 19.5%, with a high of 37.5% for TSL generator and a low of 10% for sed.

As a contrast, replicas killed 79.4%, 41.4% and 63.3% of all extra mutants for TSL generator, sed and gzip. (Recall that all the mutants killed by grep’s field executions are also killed by its in-house tests.) On average, replicas killed 61.4% extra mutants, which is 3.14 times of the percentage of extra mutants killed by inputs generated by the baseline technique, Klee.

For replicas generated for each TSL generator field executions, I also computed the number of mutants they killed and compared the set of killed mutants with the set killed by their originals as an additional experiment to show the usefulness of replicas. I also

compared the killed mutants with the original in-house tests. The average results are shown in Table 5.3.

Table 5.3: Mutants killed by individual replicas.

<b>Avg. (R)</b>	<b>Avg. (F)</b>	<b>Diff<sub>R-T</sub></b>	<b>Diff<sub>F-T</sub></b>	<b>Diff<sub>R-F</sub></b>
100.26	176.85	8.69	8.13	29.18

*Avg. (R)* and *Avg. (F)* means the average number of mutants killed by each replica and each field execution, respectively. *Diff<sub>A-B</sub>* stands for number of mutants killed by A but not B. *R* refers to replicas, *F* refers to field executions and *T* means in-house tests. Note that *R* and *F* both consider a single generated test or execution, while *T* considers all the in-house test cases.

On average, each generated test kills 100.26 mutants and it is fewer than the average number of mutants killed by a field execution. This is as expected because Replica does not necessarily reproduces the exact field execution; instead it mimics segments of the execution that manifest untested behavior. Therefore the generated execution is likely to be shorter than its original and thus kills fewer mutants.

Both replicas and their originals kill mutants that are not killed by in-house tests. On average, each replica test killed 8.69 extra mutants and this is slightly more than each field execution can kill (8.13). In addition, a single replica kills 29.18 extra mutants on average that its original does not kill. Figure 5.4 shows the breakdown for the comparison of mutants killed by each replica and its original.

Each bar in Figure 5.4 represents the number of mutants killed by a single replica and they are in a descending order. For each replica, the blue part of each bar presents the number of mutants also killed by its original and the orange part presents the number of mutants that the original does not kill. The figure shows that a large portion of mutants killed by a replica are also killed by its original as a replica is supposed to mimic its original to some extent. However, due to the fact that Replica has a certain degree of flexibility when mimicking the field execution, the generated test may not follow the exact path to exhibit

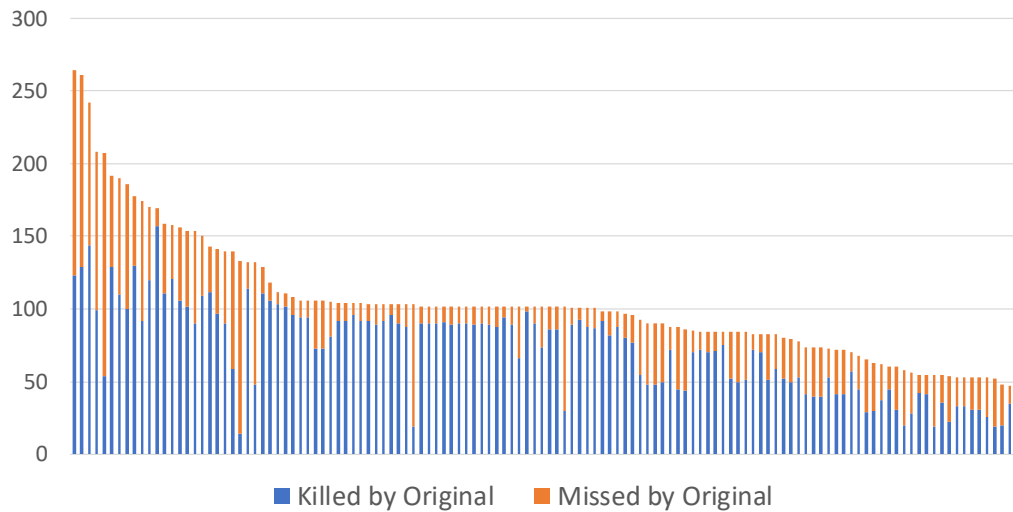


Figure 5.4: Mutants killed by individual replicas.

the same untested behavior. Even if a replica takes the same path as its original, their inputs are not identical. Both different paths and different inputs lead to killing different sets of mutants.

To summarize, the results prove that replicas can kill most mutants that are killed by field executions but not killed by in-house tests. Each replica not only kills some mutants that its original field execution kills, but also kills extra mutants that the field execution does not kill.

### 5.3.5 Limitations and Threats to Validity

The main limitation of Replica is that its underlying technique is symbolic execution, especially because of the cost of constraint solving. However, researchers have been working on various techniques to reduce the cost of constraint solving ([15], [61], [62]). These techniques show promising results and make it possible to scale symbolic execution to large systems.

Another limitation of Replica is that it does not generate test oracles. However, there

are several ways the generated inputs can be leveraged. First, some generated inputs can cause the program to crash and throw exceptions and they can be directly by developers to debug the program. Second, other inputs can be presented to the developers to examine. Third, the generated inputs can be used in regression testing. Note that most of the generated inputs focus on the correctness of the program, but some inputs can also be used to reveal non-functional aspects such as performance bugs.

There are some threats to validity of the results. Internally, there are potential mistakes in the process of running experiments and analyzing the results. To reduce the threat, I carefully inspected some results randomly sampled from the experiment and manually checked their correctness. Externally, I only considered four subject programs in our evaluation due to the difficulty of obtaining field execution information. My results may not generalize to other systems. However, the programs I used are real-world programs that come with real in-house tests and are widely used by field users. The field data used in my study is collected from real users using these programs in real-world scenarios.

## CHAPTER 6

### FURTHER IMPROVEMENT ON TEST GENERATION

In this chapter, I present a technique, Value-Aided Symbolic Execution (VASE), that aims to improve the effectiveness and efficiency of symbolic execution by leveraging observed values from user executions.

#### 6.1 Motivation

As discussed in Chapter 5, my technique, Replica, that generates tests to mimic user executions is based on symbolic execution and mitigates the path explosion problem in symbolic execution by using an incremental, guided search approach. However, symbolic execution has another inherent limitation, that the cost of constraint solving gets extremely expensive as symbolic execution explores deeper program states and builds more complex constraints. Solving a constraint on a finite domain is an NP-complete problem in general [63]. An underlying solver being unable to solve a constraint within a reasonable time stops symbolic execution from exploring deeper and more meaningful program states. In fact, after analyzing the evaluation results of Replica, I observed that symbolic execution fails to generate inputs that exercise complicated execution paths due to the fact that the underlying solver is unable to solve the collected path conditions for such paths. Even when the solver succeeds in finding a satisfying solution, the time used in constraint solving still affects the effectiveness of symbolic execution. Therefore, to improve the effectiveness of symbolic execution, we have to reduce the cost of constraint solving.

Researchers have explored techniques that target various aspects of constraint solving to reduce its cost. The satisfiability modulo theories (SMT) problem [64] is a decision problem for logical formulas with respect to some logic theory. Typical theories of interest such as linear and non-linear arithmetic, bit vectors, arrays, and so on have been extensively studied

and applied in solvers such as CVC4 [65], STP [66], Yices [67] and Z3 [68]. In addition, caching and reusing the results of previous solved constraints has been proposed to reduce the number of solver calls in symbolic execution. KLEE [10] caches counter examples to reduce the number of queries that need to be passed to the solver in the same execution. Green [15] slices and canonicalizes constraints before storing them so that the results can be reused for different programs across different executions. However, the caching mechanism helps only when the solver successfully finds a solution; it does not help in the case where the constraint is too complex to solve in the first place.

To address this problem, I have developed Value-Aided Symbolic Execution (VASE), a technique that can reduce the cost of constraint solving in symbolic execution by leveraging values observed from user executions. VASE takes as input a program under analysis and works as follows. It first runs an instrumented version of the program and logs the concrete values for variables at specific program points. Then it analyzes the collected values for each variable to identify properties of the observed values. VASE starts input generation using traditional symbolic execution at first. At each program point where a constraint needs to be solved, VASE checks whether any variable at this point have inferred properties. If so, VASE uses the properties to simplify the constraint before passing it to the underlying solver. After the simplified constraint is solved, VASE continues the symbolic execution. I present the design, implementation and evaluation of VASE in the rest of this chapter.

## **6.2 Intuition Validation**

The VASE technique is inspired by two intuitions: First, replacing some variables in the constraint with concrete values can simplify the constraint and thus reduce the time used to solve it. Second, certain program variables can take only limited values and these variables are suitable to be replaced. This section presents the evidence of the validity of the two intuitions.

### 6.2.1 The Impact of Variable Replacement

Most SMT solvers such as Z3 [62] and Yices [67] perform simplification on the constraints to solve before solving them. One common way to simplify constraints is to identify equational definitions within a context and reducing the remaining formula using the equality relationship. For example, a constraint  $(x = 4) \wedge f(x)$  will be simplified as  $f(4)$ . In this way, replacing some variables with concrete values can help solvers to simplify constraints early and thus reduce the cost of constraint solving.

To investigate how much variable replacement affects constraint solving, I randomly selected 1000 constraints from SMT-LIB benchmarks <sup>1</sup> and solved each constraint using Z3 with a timeout of 10 minutes. I logged each constraint’s satisfiability, solving time as well as the solution if the constraint is satisfiable.

Table 6.1: SMT-LIB benchmarks.

avg. variable #	avg. assertion #	solver result			
		timeout	unknown	unsat	sat
748	21	56.8%	1.5%	13.9%	27.8%

Table 6.1 shows the basic information and satisfiability of the selected constraints. On average, there are 748 variables and 21 assertions per constraint. More than half (56.8%) of the selected 1000 constraints timed out and 1.5% returned *unknown* (likely due to incomplete solving procedures or theories in Z3). 13.9% of the constraints are unsatisfiable and 27.8% can be satisfied.

Among 27.8% constraints that Z3 can find solutions for, 82 were solved in more than 1 minute. For each of the 82 constraints, I randomly selected one variable used in the constraint and replaced it with its concrete value from the logged solution. By doing so, I get a slightly simplified version of the original constraints. Finally, I reran Z3 on the simplified constraints and logged the time used to solve them. Note that Z3 applies some heuristics and there is some randomness in the solving procedure. The solver time for the

<sup>1</sup><http://smtlib.cs.uiowa.edu/benchmarks.shtml>

same constraint may differ during different runs. I used Z3 to solve the original and the simplified constraints 5 times and compared their average solver time.

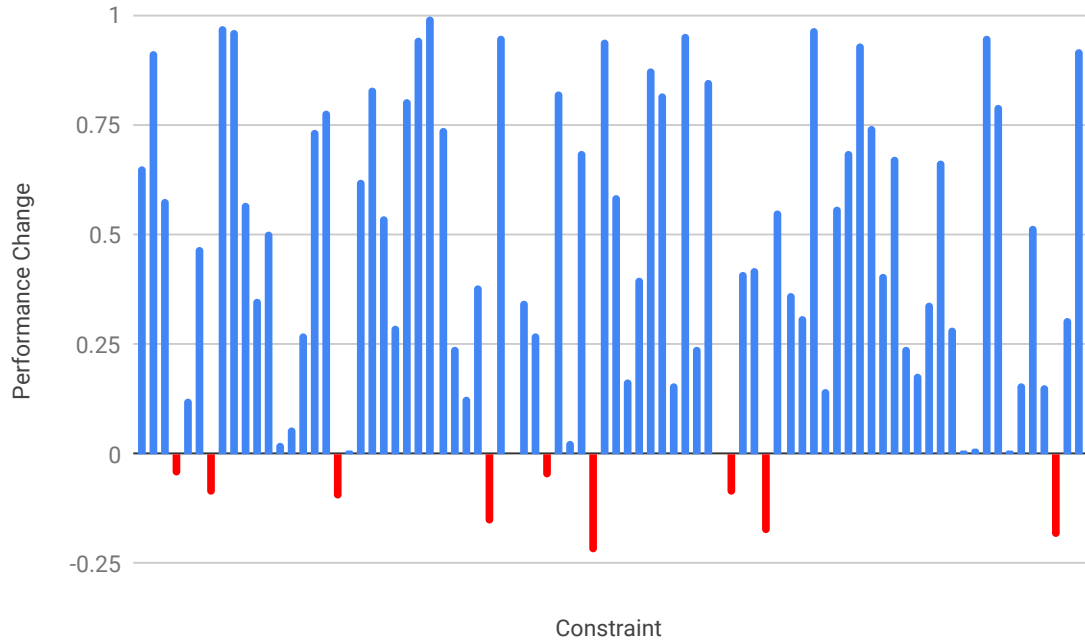


Figure 6.1: Performance change after variable replacement.

Figure 6.1 presents solver performance change after a random variable is replaced by a satisfying concrete value. The x-axis presents each individual constraint and the y-axis shows the performance change, computed using the following equation:

$$Performance\ Change = \frac{time_{original} - time_{simplified}}{time_{original}} \quad (6.1)$$

$time_{original}$  and  $time_{simplified}$  represent the average time Z3 used to solve each constraint before (*original*) and after (*simplified*) the variable replacement. As shown in the figure, Z3 used less time to solve the simplified constraint for most (87%) constraints. For about 22% of all the constraints, Z3 used 80% less time to solve the simplified ones. The average solver time saved by replacing one variable in a constraint is 43%. The  $p$ -value for the paired  $t$ -test is less than 0.0001, indicating that the time difference in solving constraints

before and after replacing a variable is considered to be extremely significant.

## 6.2.2 Limited-Valued Variables

As shown in Section 6.2.1, replacing variables in a constraint with a satisfying concrete value can reduce the solver time significantly. However, a question remains: what variables can be replaced by concrete values without over constraining the search space for symbolic execution? A constraint can contain a large number of variables and not all of them are suitable to be replaced. I examined some program variables in concrete executions and observed that these variables can be categorized into two types. The first type of variables have a large number of values from a rich domain in different executions. The second type of variables have a fixed set of values that are observed at the same program point across a large number of executions, regardless of the inputs. In the following of this chapter, I call the second type of variables *limited-valued variables*.

```
public void parse(String input) {
1.  int idx1 = input.indexOf("<");
2.  int idx2 = input.indexOf(">", idx1);
3.  if (idx1 != -1 && idx2 != -1 ) {
4.      String content = input.substring(idx1 + 1, idx2);
5.      if (content.startsWith("/"))
6.          String property = content.substring(1);
7.          if (property.equals("li")) {
8.              //do something
9.          else if(property.equals("dt")) {
10.             //do something
11.         }
    }
}
```

Figure 6.2: Simple code example to limited-valued variables.

Consider the program that parses an input string, shown in Figure 6.2. The program first gets the sub-string of the input between "`<`" and "`>`" and checks whether the sub-string starts with the character `/`. To execute the true branch in line 6, Symbolic execution creates

a constraint:

$$\begin{aligned}
& sym\_input.indexof("<", 0) \neq -1 \wedge \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0))) \neq -1 \wedge \\
& sym\_input.substring(sym\_input.indexof("<", 0) + 1, \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0)))) \text{.startswith("/")} = true
\end{aligned} \tag{6.2}$$

Assume the solver has problem solving the constraint and continuing to explore the true branch. We want to use observed values to simplify the constraint. After executing the program with different inputs, we observe that variable *content* at line 5 always take a fixed set of values: *"/li", "/dt"* when the executions take the true branch. By analyzing the program itself, we find that this observation holds true for all executions. We can leverage the observation and use *"/li"* (or *"/dt"*) to replace *content*. The constraint will be simplified to

$$\begin{aligned}
& sym\_input.indexof("<", 0) \neq -1 \wedge \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0))) \neq -1 \wedge \\
& sym\_input.substring(sym\_input.indexof("<", 0) + 1, \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0)))) = "/li" \wedge \\
& "/li" \text{.startswith("/")} = true
\end{aligned} \tag{6.3}$$

And it will be further simplified to

$$\begin{aligned}
& sym\_input.indexof("<", 0) \neq -1 \wedge \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0))) \neq -1 \wedge \\
& sym\_input.substring(sym\_input.indexof("<", 0) + 1, \\
& sym\_input.indexof(">", (sym\_input.indexof("<", 0)))) = "/li"
\end{aligned} \tag{6.4}$$

The result of the modification is that a relatively complex operation (*startswith*) is replaced by an equality relationship, which is easier for the solver to solve.

Based on the observation that the observed values for limited-valued variables can reflect properties of the program or its variables, these values can be leveraged to provide extra information to simplify constraints in symbolic execution. Therefore, my proposed solution is to replace limited-valued variables with their observed values.

To investigate the existence of limited-valued variables, I performed a preliminary study to track the values for program variables in the program *pdfbox*<sup>2</sup> at each conditional statement and counted the number of observed values for each variable that appear a large number of executions. The result shows that, there are 1227 variables that appear in more than 100 executions at the same program point, among which 308 (26%) non-boolean variables have no more than 10 observed values.

A manual investigation shows that the values that a variable can have are limited to a small number in two scenarios. The first scenario is that the values of a variable is constrained by the operation such as modulo and signage. Intuitively, if a symbolic variable can only have a few satisfying values, it is hard for the solver to find such a value. Using an observed value for the variable can help solve the problem. The second scenario is that the values of a variable are confined by the domain. The symbolic variable in the constraint can take a large number of values, but the program restricts the values to a smaller domain. In this case, the observed value can help reduce the search space for the constraint solver. The preliminary result shows that limited-valued variables exist and their existence can be leveraged to reduce the cost of constraint solving.

### **6.3 An Approach for Improving Symbolic Execution Using Observed Values**

In this section, I will present VASE (Value-Aided Symbolic Execution), a technique that leverages the observed values from field executions to improve the effectiveness and effi-

---

<sup>2</sup><http://www.pdfbox.org/>

ciency of symbolic execution.

### 6.3.1 Overview

Figure 6.3 shows a high-level overview of VASE. VASE has three components: an *Instrumenter*, a *Value Analyzer* and an *Input Generator*. Given a program  $P$ , *Instrumenter* instruments  $P$  such that when run, it records the observed values for variables  $V$  at specified program points. *Value Analyzer* takes in the collected values for each variable in  $V$  and infers properties from them. Finally, the *Input Generator* incorporates the inferred properties to generate test inputs using symbolic execution. The rest of the section describes each component in detail.

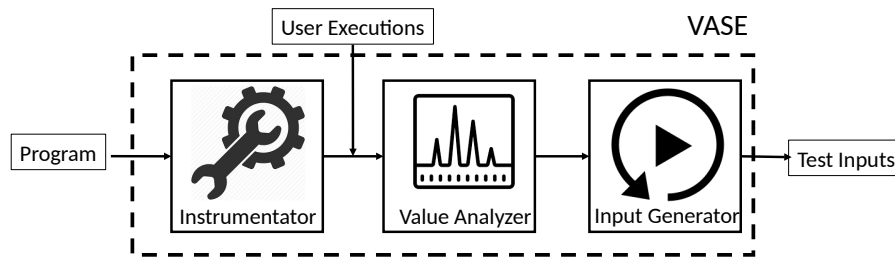


Figure 6.3: High-level overview of VASE.

### 6.3.2 Instrumenter

To collect observed values for the variables at certain program points, VASE uses a component, *Instrumenter* to insert probes at each conditional statement  $S$ . During the execution of the instrumented program  $P$ , if a conditional statement  $S$  is encountered and  $S$  has a probe inserted,  $P$  logs the location of  $S$ , variables that need to be tracked and their observed values during this execution. For variables of primitive types or of string type, their values are logged directly. Variables of reference types are traversed recursively until the values for

all fields of primitive types or string type are logged. A *visited* flag is associated for each reference typed variable to avoid redundant visit in case of recursive reference.

### 6.3.3 Value Analyzer

After VASE collects observed values from a number of concrete executions, it uses a *Value Analyzer* to categorize and infer properties from these collected values. It first consolidates the values observed for the same variable at the same program point for all concrete executions and counts the number of occurrences of the variable at the program point. At each program point where one or more variables' values are logged, it then processes each variable *var* using the procedure described in Algorithm 4.

<p><b>Input:</b> <math>concValues = \{(S, \{(var, \{val\})\})\}</math>, which maps a program point <math>S</math> to a set of variables <math>\{var\}</math> tracked at <math>S</math> and <math>var</math>'s observed values <math>\{val\}</math>  <i>minOccurrence</i>, the minimum number of times that a variable occurs at a program point  <i>maxValue</i>, the maximum number of concrete values that are observed for a variable at a program point</p> <p><b>Output:</b> <math>propertyMap = \{(S, \{(var, \{prop\})\})\}</math>, which maps a program point <math>S</math> to a set of variables <math>\{var\}</math> tracked at <math>S</math> and inferred properties <math>props = \{prop\}</math> for <math>var</math></p> <pre> 1: <math>propertyMap \leftarrow \emptyset</math> 2: <b>for</b> <math>(S, varMap)</math> in <math>concValues</math> <b>do</b> 3:   <b>for</b> <math>(var, vals)</math> in <math>varMap</math> <b>do</b> 4:     <b>if</b> <math>var.count &lt; minOccurrence</math> <b>then</b> 5:       <b>continue</b> 6:     <b>end if</b> 7:     <b>if</b> <math>vals.size() \leq maxValue</math> <b>then</b> 8:       <math>varProperty.put(var, vals)</math> 9:     <b>else if</b> <math>var</math> is a string variable <b>then</b> 10:      <math>prop\_len = common\_length(vals)</math> 11:      <math>prop\_charat = common\_charat(vals)</math> 12:      <math>varPropertyMap.add(prop\_len, prop\_charat)</math> 13:    <b>end if</b> 14:    <math>propertyMap.put(S, varPropertyMap)</math> 15:  <b>end for</b> 16: <b>end for</b> 17: <b>return</b> <math>propertyMap</math> </pre>
---

**Algorithm 4:** ValueAnalysis

The algorithm takes in the consolidated concrete values *concValues* that map a program point  $S$  to a set of variables  $V = \{var\}$  tracked at  $S$  and each  $var$ 's observed values  $vals = \{val\}$ . The variable  $var$  also contains extra information on the total occurrence of  $var$  at  $S$  across executions ( $var.count$ ). The inputs to the algorithm also include two customizable thresholds: *minOccurrence* for identifying variables that occur a large number of times and *maxValue* for specifying the maximum number of values a variable can have to be

considered as a limited-valued variable. The output of the algorithm is a set of properties inferred for variables being tracked at each program point.

*ValueAnalysis* first initializes *propertyMap* to a empty set (line 1) and then iterates over all the program points where variables are tracked. At each program point  $S$ , if a tracked variable  $var$  occurred less than *minOccurrence* times during all concrete executions,  $var$  will not be considered as a candidate for value replacement (lines 4–6). Otherwise, the algorithm counts the number of values observed for  $var$ . If the number is no more than the threshold *maxValue*,  $var$  is considered as a limited-valued variable and its observed values *vals* are stored (lines 7–8).

Due to the special characteristics of String, further properties are considered for variables of String type if the string variable  $var$  is not limited-valued. These properties include the common length of all observed values for  $var$  (-1 as default if the lengths are not the same) and the same character at the same position of all observed string values. These string properties represent equality relationships and can be utilized by solvers to simplify constraints. If these properties are not default values, they are added to the property map for  $var$  (lines 9–13). After analyzing values for all the tracked variables in  $S$ , the *variable – property* map and its location  $S$  is stored in the *propertyMap* (line 14). The algorithm returns the *propertyMap* when the analysis is done for all program point  $S$  (line 17). The stored properties are used to provide extra information for constraint during input generation presented in Section 6.3.4.

#### 6.3.4 Input Generator

The final component of VASE is an *Input Generator* and Figure 6.4 illustrates its main work flow.

The same as classic symbolic execution, VASE’s *Input Generator* executes the program with symbolic inputs and keeps track of the symbolic states at any program point. When a branch statement is executed, it constructs the constraint by adding the path condition used

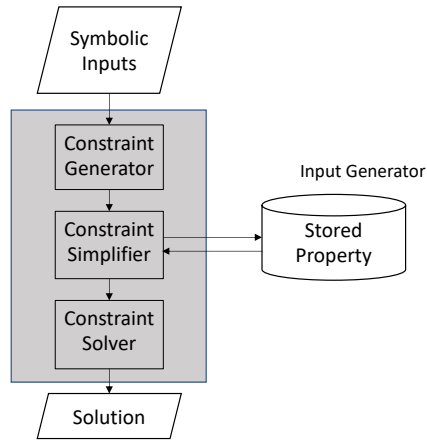


Figure 6.4: Input Generator.

in the condition, as in classic symbolic execution. Then, instead of consulting the solver directly for the constraint’s satisfiability, *Input Generator* first checks whether there are stored properties for current branch statement. If yes, it modifies the constraint by leveraging the property. Finally it passes the modified constraint to the underlying constraint solver to ask for a solution. Algorithm 5 presents the flow in which VASE incorporates the stored properties to modify constraints.

The algorithm takes in the *constraint* generated at a branch statement  $S$  during normal symbolic execution and the *propertyMap* that stores the properties inferred using Algorithm 4. The purpose of the algorithm is to leverage the properties to modify the original constraint to make it easier for the underlying solver to solve.

If there are properties inferred for variables at the current branch statement  $S$ , it performs the constraint simplification (lines 1–50). *ConstraintModification* first creates an empty set *triedValue* to record the variables and their properties that fail to make the original constraint satisfiable (line 2). The flag *success* indicates whether a property is successfully selected to modify the constraint and it is initialized to *false*. The algorithm retrieves

**Input :** *constraint*: The original constraint generated by symbolic execution  
*S*: The branch statement where the constraint is generated  
*propertyMap* =  $\{(S, \{(var, props)\})\}$ , which maps a program point *S* to a set of variables *var* tracked at *S* and inferred properties *props* for *var*

```

1: if S in propertyMap then
2:   triedValue  $\leftarrow$   $\emptyset$ 
3:   success  $\leftarrow$  false
4:   varPropMap  $\leftarrow$  propertyMap.get(S)
5:   oriConst  $\leftarrow$  constraint
6:   curConst  $\leftarrow$  constraint
7:   for var in varPropMap.vars() do
8:     if var is not symbolic then
9:       continue
10:    end if
11:    props  $\leftarrow$  varPropMap.get(var)
12:    while true do
13:      prop  $\leftarrow$  selectProp(props, triedValue)
14:      if prop == null then
15:        if curConst! = oriConst then
16:          triedValues.remove(var)
17:          var  $\leftarrow$  previous variable
18:          continue
19:        else
20:          break
21:        end if
22:      end if
23:      if isValue(prop) then
24:        simplify(constraint, var, prop)
25:      else
26:        newCons  $\leftarrow$  createConst(var, prop)
27:        addConstraint(constraint, newCons)
28:      end if
29:      result  $\leftarrow$  solver.solve(constraint)
30:      if result == UNSAT then
31:        triedValue.add(prop)
32:        constraint  $\leftarrow$  curConst
33:        continue
34:      else if result == SAT then
35:        success  $\leftarrow$  true
36:        break
37:      else
38:        curConst = constraint
39:        if !isValue(prop) then
40:          triedValue.add(prop)
41:        continue
42:        end if
43:        break
44:      end if
45:    end while
46:    if selected then
47:      break
48:    end if
49:  end for
50: end if

```

**Algorithm 5:** ConstraintModification

the variables and their associated properties from the storage (line 4). It also makes a copy of the original constraint so that it can be restored later (line 6). It also tracks the modified constraint that is not unsatisfiable with *curConst*. For each variable *var* that has inferred properties, if it is not a symbolic value, the algorithm continues to select

a different variable (line 9). Otherwise the algorithm gets the properties associated for *var*. The modification loop starts by randomly selecting a property of *var* that is not in *triedValue* (line13). If no property is selected, the algorithm checks whether the constraint has been already modified, indicating whether a previous variable has been replaced. If so, it backtracks to the previous variable to try a different property and removes *var* from *triedValues* because the values tried for *var* with a different value for the previous variable can make the constraint satisfiable. Otherwise, the next variable is tried (lines 14 –22). If a property is selected, the algorithm checks whether the property is a concrete value or a string property. If it is a concrete value, the original *constraint* is simplified by replacing *var* with the concrete value and performing basic constant folding operations (lines 23–24). The operations are performed with a post-order traversal on the tree representing the constraint. Otherwise it creates a new constraint corresponding to the string property. The newly created *newCons* is then added to the original constraint in order to provide extra information to the solver (lines 25–27). VASE leaves simplification of string properties to the solver because it is often that case that these properties are not explicitly used in the original constraint.

After modifying the existing constraint, the algorithm passes the new constraint to the solver to check its satisfiability (line 29). If the new constraint cannot be satisfied, it is likely that the selected property conflicts with the original constraint. And thus the property is discarded for the current attempt but added to the *triedValue* set so that it will not be selected again for the same modification. The original constraint is restored to try the next property (lines 30–33). The *success* flag is set to true to signal the termination of the modification. (lines 34–36 and line 47). However, if the solver returns neither “UNSAT” or “SAT”, it means the solver times out for the new constraint. In this case, the algorithm checks again whether the selected property is a concrete value. If not, it selects a different property to add to the current variable because the string properties inferred for the same variable do not contradict each other. Otherwise, it selects a different variable and continues the modification (lines 37–43). Note that the algorithm adds a variable in this case instead

of discarding the original modification because we assume that the constraint requires extra information if the solver fails to solve it within the given time.

Note that the observed properties are only used to simplify the constraint passed to the solver to determine the satisfiability and are not used to change the path condition for later execution. Therefore, VASE does not give up any exploring execution path that would be explored by classic symbolic execution.

## 6.4 Evaluation

To evaluate the effectiveness of VASE, I investigated the following research questions.

- **RQ1: Can VASE help reduce the cost of constraint solving in symbolic execution?**
- **RQ2: Can VASE generate inputs that cover more code than inputs generated by classic symbolic execution?**

To address the above questions, I implemented the technique described in Section 6.3 and applied it to four real world library systems.

### 6.4.1 Implementation

The VASE technique is implemented to work on Java programs and the implementation consists of three components (as shown in Figure 6.3).

The *Instrumenter* is an extension of Java PathFinder (JPF)<sup>3</sup>. JPF is a system to verify executable Java bytecode programs and it provides a variety of mechanisms to extend the system for purposes like gathering execution statistics, monitoring program states and so on. VASE's *Instrumenter* leverages the mechanism and extends JPF through a customized listener that dumps values of variables whenever a conditional statement is executed.

The *Value Analyzer* is implemented using Python scripts to merge values collected from multiple concrete executions and process them using Algorithm 4.

---

<sup>3</sup><http://javapathfinder.sourceforge.net/>

The *Input Generator* uses the Symbolic PathFinder (SPF) <sup>4</sup>, which leverages JPF’s analysis engine to perform symbolic execution on Java bytecode. VASE extends SPF by overloading its *execute()* method for conditional instructions based on the process described in Algorithm 5.

#### 6.4.2 Subject Programs

To answer the research questions in a real-world setting, I used four non-trivial Java programs as evaluation benchmarks: jodd, jsoup, pdfbox and xstream. I selected these four programs because they are widely used library systems. More importantly, they all take in non-trivial inputs and process the inputs with complex logics. Running symbolic execution on these subjects are likely to produce complicated constraints. Table 6.2 shows the basic information about the benchmark programs.

Table 6.2: Benchmarks used to evaluate VASE.

<b>Program</b>	<b>Description</b>	<b>Size</b>	<b>Tests</b>
<b>jodd</b>	Lightweight Java utilities	38k	414
<b>jsoup</b>	HTML parser	18k	747
<b>pdfbox</b>	PDF manipulation	140k	454
<b>xstream</b>	XML parser	35k	1579

Column *Description* briefly summaries the main purpose of the each program. Column *Size* shows the size of each program in lines of code (LOC). Column *Tests* refers to the number of unit test cases that come with each program. The size of the four benchmarks ranges from 18 kLOC to 140 kLOC and the programs come with between 414 and 1579 unit test cases. The number of unit test cases matters in the evaluation because the provided test cases can (1) serve as concrete executions to collect observed values from, and (2) be converted as execution stubs to start symbolic execution.

<sup>4</sup><https://github.com/SymbolicPathFinder/jpf-symbc>

### 6.4.3 Experiment Setup

With the four benchmarks, I set up the experiment to compare VASE with Green [15]. I chose Green as the baseline technique because it is also built on top of Symbolic PathFinder and its goal is to reduce the cost of constraint solving by caching and reusing constraints.

#### *Experiment Protocol*

The experiment is conducted as follows:

1. *Value Collection*: To collect observed values for the benchmark programs, I ran unit test cases against the instrumented programs and used them as concrete executions to observe values from.
2. *Value Analysis*: I then used *Value Analyzer* to aggregate concrete values, infer properties from them and then store these properties in a file.
3. *Symbolic Execution*: I used each unit test case as an execution to run symbolically and ran them using VASE and Green, with a 30-minute timeout for each execution and 30-second timeout for each solver call. The underlying solver I used is Z3 [62].
4. *Result Comparison*: I compared the effectiveness of VASE and Green using the following statistics: the number of timeout constraints, the average time for each solver call, the number of solver calls and the code covered by each technique.

#### *Stub Generation*

Both Java PathFinder and Symbolic PathFinder require a configuration file that specifies the main class as an entry point to start an execution. However, not all Java programs, especially in library systems, contain a class with the main method. In addition, Symbolic PathFinder requires an explicit configuration for the method to be executed symbolically, one for each execution/configuration. Manually writing the configuration file for each method is

a time-consuming task. Therefore, I implemented an assisting tool to automatically generate execution stubs.

The stub generation tool iterates over all the unit test cases and wraps each test case with a main method that uses *JUnitCore* to run the test case. It also generates a configuration file for each test case to specify the generated wrapper as the main class. The stubs are generated for both the value collection step and the symbolic execution step described above.

For the symbolic execution step, the generated configuration file also identifies the method tested in each test case and marks it as symbolic.

#### 6.4.4 Results and Discussion

*RQ1: Can VASE help reduce the cost of constraint solving in symbolic execution?*

Table 6.3: Constraint Solving Performance.

	<b>Benchmark</b>	<b>Baseline</b>	<b>VASE</b>	<b>Improvement</b>
<b>Timeout constraints</b>	jodd	53	25	52.8%
	jsoup	32	10	68.8%
	pdfbox	67	25	62.7%
	xstream	16	9	43.8%
<b>Avg. solver time (ms)</b>	jodd	188	86	54.3%
	jsoup	1100	818	26.3%
	pdfbox	22	20	11.9%
	xstream	65	41	36.8%
<b>Solver calls</b>	jodd	9738	5765	40.8%
	jsoup	732	438	40.2%
	pdfbox	1085	416	61.7%
	xstream	92	26	71.7%

Intuitively, the observed values can help reduce the cost of constraint solving in two ways. First, it can help the solver solve constraints that it originally fails to solve due to the time limit. Second, it can reduce the time used to solve the constraints that can be solve originally. Therefore, I evaluated the performance of constraint solving in the two aspects: the number of unique constraints that the underlying solver failed to solve within the given time limit (*Timeout constraints*), and the average time used to solve each individual

constraint (*Avg. solver time*). In addition, I considered the average number of invocations made to the underlying solver in each execution (*Solver calls*).

Table 6.3 shows the performance in the above three aspects for symbolic execution performed by the baseline technique and VASE, respectively. The results are presented for each of the benchmark program. It also presents the improvement VASE brought compared with the baseline technique computed using Equation 6.5.

$$Improvement = \frac{Metric_{baseline} - Metric_{VASE}}{Metric_{baseline}} \quad (6.5)$$

*Metric* refers to the number of timeout constraints, the average solver time for individual constraints and the number of solver calls.

**Timeout constraints.** For the baseline technique, the numbers of timeout constraints range from 16 (for *xstream*) to 67 (for *pdfbox*). With the help of observed values, VASE reduces these numbers by 52.8%, 68.8%, 62.7% and 43.8%, with an average of 57.0%. For each benchmark, a large number of executions encounter constraints that the underlying solver fail to solve within the given time limit (30 seconds). However, since the executions use unit test cases as stubs, some executions end up executing the same parts of the program and sharing the same constraints that the solver fail to solve. Therefore, for all four benchmarks, the number of unique constraints that timed out is relatively small.

**Average Solver Time.** The second criterion I used to evaluate the performance of constraint solving is the average time used by the solver to solve an individual constraint. With the baseline technique, the underlying solver takes 188ms, 1100ms, 22ms and 65ms for each individual constraint generated by programs *jodd*, *jsoup*, *pdfbox* and *xstream* respectively. VASE reduces the solver time to 86ms, 818ms, 20ms and 41ms for each program. The decrease in terms of solver time used for each individual constraint is 54.3%, 26.3%, 11.9% and 36.8%, with an average of 32.3%. Note that most constraints during symbolic execution are solved within 100ms and the performance improvement caused by VASE for these constraints is not obvious. However, for constraints that took a longer time

to solve or the solver failed to solve, VASE reduced the solver time significantly.

**Solver Calls.** The number of solver invocations for each execution vary a lot across different benchmarks. On average, each *xstream* execution calls the underlying solver the least (92 times on average) while each *jodd* execution calls the solver the most (9738 times on average). Most of the solver invocations for *jodd* happen for executions involving formatting functions which perform conditional checking frequently. These executions invokes the underlying solver more than 300k times, making the average solver invocation for *jodd* much higher than other benchmarks.

VASE reduces the numbers of solver calls to 4541, 438, 416 and 26 for *jodd*, *jsoup*, *pdfbox* and *xstream*, respectively, with an average decrease of 53.6%. The reason the VASE can reduce the number of solver calls is that VASE helps the underlying solver to solve the constraints that it failed to solve originally and the constraints are then cached. These cached constraints are reused to avoid invoking the underlying solver redundantly.

*RQ2: Can VASE help improve code coverage?*

Both JPF and SPF work on Java bytecode and calculate statistics per execution. Therefore, I compare the number of application instructions covered by each execution when run with the baseline technique and VASE. If an execution covers more instructions with VASE than with the baseline technique, I call it an improved execution.

Table 6.4: Percentage of improved executions

<b>Benchmark</b>	jodd	jsoup	pdfbox	xstream
<b>Improved executions</b>	25.8%	22.5%	11.7%	13.2%

Table 6.4 presents the percentage of executions that are improved by VASE for each benchmark. The numbers show that compared with the baseline technique, VASE increased the number of covered instructions in 25.8% executions for *jodd*, 22.5% for *jsoup*, 11.7% for *pdfbox* and 13.2% for *xstream*. I performed a further investigation to understand why a large number of executions are not improved by VASE and found two reasons. First, some

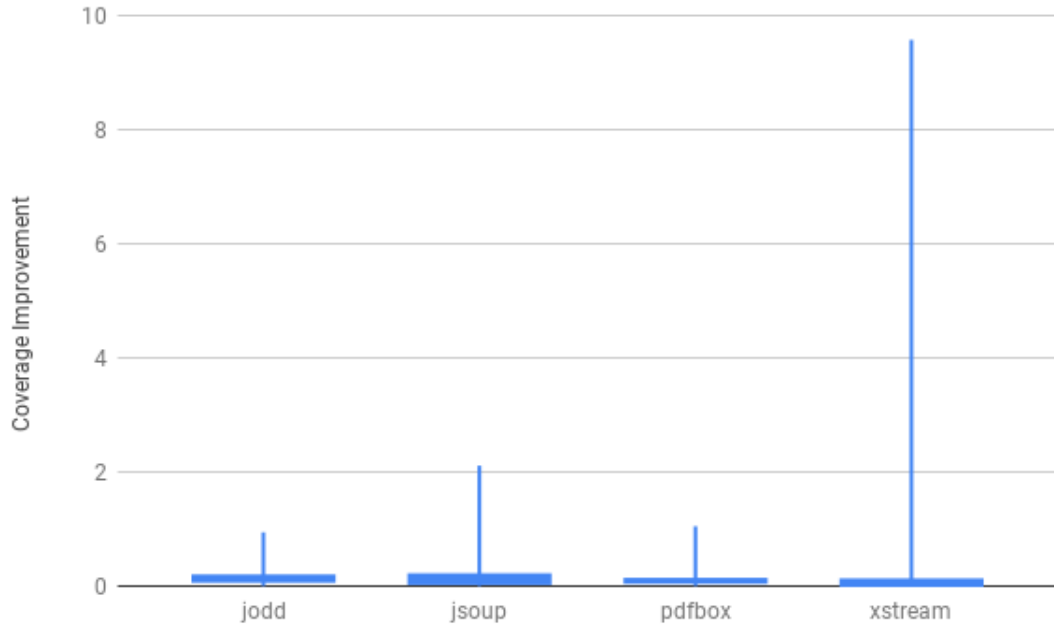


Figure 6.5: Coverage Improvement

executions do not encounter statements with inferred properties at all. As a result, VASE is not applicable for these executions. Second, symbolic execution finishes the search for some executions before it hits the predefined time limit and leaves no extra instructions to cover by VASE.

For executions whose instruction coverage is improved by VASE, I calculated the improvement for each of them using the following equation, where  $covered_{VASE}$  and  $covered_{baseline}$  refer to the number of instructions covered by VASE and the baseline technique, respectively.

$$Coverage\ improvement = \frac{covered_{VASE} - covered_{baseline}}{covered_{baseline}} \quad (6.6)$$

Figure 6.5 is a box plot that presents the improvement on the covered instructions for each execution. The medians are 10%, 16%, 8% and 3% for *jodd*, *jsoup*, *pdfbox* and *xstream*, respectively. I examined these improved executions in detail and found that VASE helps increase the number of covered instructions due to two reasons. First, VASE helps the solver

to solve previously unsolvable constraints. As a result, symbolic execution can explore more program paths and cover more instructions. Second, VASE reduces the time used to solve individual constraints and also reduces the number of solver calls. The time used in the underlying solver is reduced significantly and symbolic execution has more time to explore more program states.

Although for most executions, the coverage improvement is less than 20%, for some executions, especially in the case of *xstream*, VASE increased the number of covered instructions by more than 9 times. I manually inspected these executions and found that there are constraints that the underlying solver failed to solve at the early part of the symbolic execution, making it difficult to cover a large portion of the program. By incorporating extra information observed from concrete values, VASE made the constraints solvable and increased the chance to cover previously uncovered instructions.

#### 6.4.5 Limitations and Threats to Validity

As in most studies, there are threats to the validity of my experiment results. Internally, there are potential mistakes in the process of implementing the technique, running experiments and analyzing the results. To reduce the threat, I carefully inspected some results randomly sampled from the experiment and manually checked their correctness. In terms of the external validity, the programs I used in the evaluation are all Java libraries and the results may not generalize to other types of programs. However, the benchmarks I considered are real-world programs with considerable sizes. They are widely used by real users in the field. Another potential issue with my experiment is that I used unit test cases provided by the each benchmark as concrete executions to collect values from. User executions in the field can be different from test executions in terms of generating observed values. Field executions can produce a larger amount of concrete values and thus making the properties inferred from them more accurate. Therefore, I expect the technique to be more robust if concrete values observed from user executions are used.

## **CHAPTER 7**

### **RELATED WORK**

Software testing is an extremely broad topic and there is a vast range of related work in this research area. In this section, I will focus on the work that is the most closely related to the approaches in my thesis dissertation, which include research using field data for software engineering tasks, software behavioral representation, test data generation, reproducing field failures and improving constraint solving for symbolic execution.

#### **7.1 Using Field Data in Software Engineering**

Researchers have investigated for over a decade the use of field data to aid various software engineering tasks traditionally performed in-house. The Gamma project, for example, aims to leverage field data to help software maintenance tasks, such as impact analysis and regression testing [69, 70]. Elbaum and Diep have investigated the potential benefit of using field data to improve software profiling [71, 72]. Cleanroom usage testing [73, 74] uses various models to represent expected usage and compares field executions to these models. However, it requires rigorous analysis of the specification, which may not be applicable in many cases. My work is related to these techniques, as its ultimate goal is to use field data to improve in-house testing activities.

My work is also related to that of Pavlopoulou and Young, who developed a technique for collecting field data about statements covered in the field but not in-house [75]. In fact, their approach could be used in my context. Hilbert and Redmiles [76] propose an agent-based approach for collecting field usage data and feedback that can provide developers with usage- and usability-related information [76]. This information can help detect and resolve mismatches between developers' expectations and actual software use. By contrast, my effort focuses on modeling and analyzing field executions.

Bug reports can also shed light on field executions and are one of the most commonly used field data types. For example, models for fault localization and automatic retrieval of faulty files based on bug reports can aid debugging [77]. These models are at least as effective as other automated debugging techniques. However, most human-written bug reports contain limited information, which reduces their utility in practice [78]. Meanwhile, automatically generated crash reports contain rich data that can help triage reports and locate bugs. ReBucket clusters duplicate crash reports collected in the field by grouping crash reports based on call stack similarity calculated using the Position Dependent Model [79]. Similarly, CrashLocator uses call stacks from the crash reports to generate approximate crash traces by stack expansion and uses the expanded traces to locate faulty methods [80]. Both techniques require large numbers of crash reports to be effective.

## 7.2 Behavioral Representation

Code coverage has long been the standard metric for test suite quality [34, 10, 81, 40, 42, 41], and executions can be characterized using variations in coverage (e.g., via basic block vectors [82]). However, recent studies have shown that coverage may not be a great indicator of test suite effectiveness at finding faults [43], and that mutation kill scores are a better metric [44]. Still, stronger proxies for representing system behavior may be desirable, such as invariants-based descriptions of the behavior [83] or finite-state-machine-based models of the behavior [26, 27, 28, 29, 30, 31, 38, 32, 39, 3, 2, 4, 5].

There are numerous algorithms to mine temporal invariant instances [84]. For example, Javert [19] infers property specifications by composing simpler micro-patterns into larger ones, focusing on efficiency.  $N$ -grams can represent executions in terms of substrings of kernel-call or application-server-call sequences [85], which is a similar representation to kTails. I use InvariMint [3, 2] to mine behavioral properties, but my work is easily extendable to other property-mining algorithms, and advances in the richness of these algorithms are complementary and beneficial to my work. Another kind of property that my work does

not include is structural, data-value properties that relate internal program variables, often described with variable values and can encode method pre- and post-conditions, as well as class-level property types. My work can be extended to use such properties (e.g., mined by Daikon [83] from program executions), and again, advances in the inference of such properties are complementary and beneficial to my work. Combining structural and temporal properties is likely to increase the precision of behavioral difference measurement between test and field executions.

Finite-state-machine-based models that describe system behavior are similarly complementary to my work. The kTails algorithm [13] is the basis for numerous behavioral model-inference algorithms [26, 27, 28, 29, 30, 31, 32, 5]. My work uses the behavioral invariants that precisely describe the models inferred by kTails, but could be adapted to use the invariants that describe the behavioral models inferred by each of these algorithms. InvariMint [3, 2] focuses on decomposing behavioral model inference algorithms into such invariants, including for the kTails [13] and the Synoptic [86, 1] algorithms. Here, I used InvariMint to infer the behavioral invariants, which facilitates expanding the work to include other kinds of invariants. However, some model-inference techniques require richer than standard FSM models, and may not be represented precisely and completely by behavioral, temporal properties. GK-Tails [31] requires EFSMs, and RPNI [38] requires Probabilistic FSMs. The Alergia algorithm [38] cannot be easily specified using InvariMint because of reliance on transition probabilities updated dynamically during the model inference procedure.

User-specified LTL formulae of desired system behavior can be combined, checked by a model checker, and used as constraints on inferring a single behavioral model [39]. By contrast, my work does not require the user, nor the developer, to know the desired system properties, instead comparing test executions to field executions. However, it is conceivable that checking for differences between user-specified properties in test and field executions may lead to further insights. Other representations of behavior are also possible, including

UML sequence diagrams [87], communicating automata [88, 89], and symbolic message sequence graphs [90]. These behavioral representations are outside the scope of my work, but my analysis could be extended to these representations as well.

### **7.3 Test Data Generation**

Test data generation is the process of creating a set of inputs for testing software systems. Researchers have proposed a considerable number of techniques to generate test data automatically. There are three mainstream methods used by these techniques: random generation, search-based generation and symbolic-execution-based generation.

JCrasher [91] is an automatic testing tool for Java code. It generates random but type-correct inputs in an attempt to cause a Java application to crash. It identifies type information for method parameters and return values, keeps a mapping between types and some pre-set values, and generates test cases based on these values. Randoop [9] is another technique that generate tests for Java programs. It improves random testing by incorporating feedback obtained by checking whether the generated inputs lead to new and valid program states.

EvoSuite [11] uses a hybrid strategy to generate test cases with assertions for classes written in Java code. It uses a search-based approach integrating state-of-the-art techniques such as for example hybrid search [92], dynamic symbolic execution [93] and testability transformation [94].

Symbolic execution is another technique commonly used for test generation. KLEE [95] uses optimized constraint solving techniques to improve the performance of symbolic execution. It is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intense programs.

Most of these techniques start from scratch and generate inputs for initial software testing. In contrast, my technique is used after the software is deployed and the generated tests are used for test augmentation.

## 7.4 Reproducing Field Failures

Another closely related topic is reproducing field failures. BugEx [96] adopted an evolutionary approach that generates executions almost identical to a single failing test case and uses these executions to identify program facts related to the failure. The goal is to perform fault localization when there is only one failing test case. Zamfir and Candea [97] proposed execution synthesis, a technique that combines symbolic execution and static analysis to automatically generate an execution that leads to the symptoms described in a given bug report. STAR [98] combines a backward symbolic execution and a method sequence composition approach to generate unit test cases that reproduce field crashes. EvoCrash [99] proposes a guided genetic algorithm for reproducing real-world crashes. They use stack traces to guide the search to reduce search space when generating a test case that can trigger an observed crash. All the above techniques generate simple test cases while my technique mimics complete executions that can manifest complex field behavior.

Kifetew and colleagues [100] used a search-based technique to generate complex grammar-based inputs for reproducing field failures. They used genetic operators that manipulate parse-tree representations of the structure input and cost function that guides the evolution of these trees to generate inputs exercising similar traces of a failing execution. Similar to my technique, BugRedux [46] also uses guided symbolic execution as the main approach to synthesize executions that reproduce failed failures. It collects different types of dynamic execution data such as call sequences, call stacks, points of failure and complete traces. It then uses the collected information to guide the generation of test cases that crash in the same way as a field failure. Although I leverage the same underlying guided symbolic execution approach, my technique has a different goal. Instead of reproducing field failures, I aim to mimic field executions that exhibits untested behavior.

## 7.5 Improving Constraint Solving for Symbolic Execution

Constraint solving has always been known to be one of the biggest limitations in symbolic execution and researchers have explored various approaches to reduce the cost of constraint solving.

One common area that has been explored for improve constraint solving is to reuse constraint solution. Green [15] reduces constraints to standard forms and stores the standard constraints to allow for reuse of their solutions within and across executions and programs. GreenTrie [101] further improves Green based on the logical implication relations among constraints. Aquino and colleagues [102] proposed novel canonical forms to identify a large class and equivalent constraints and constraints related by logical implication. These techniques that reuse constraint solution reduce constraint solving cost by avoiding redundant solver calls for constraints already solved. My VASE technique is different in that it can help solvers to solve constraints that are originally not solved.

Erete and Orso [103] addressed the constraint solving problem using a different approach. They proposed an optimization strategy that uses domain and contextual information to eliminate irrelevant or potentially irrelevant constraints. Their technique is similar to VASE in that it also uses concrete values to replace symbolic variables. However, this technique obtains the concrete values during concolic execution while VASE collects the values from a large number of real executions. Păsăreanu and colleagues [104] also adopted the approaching of mixing concrete and symbolic solving to improve symbolic executions. They split the path condition into simple constraints and complex non-linear constraints, then use the solution for the simple constraints to simplify the complex ones. Their technique has a different goal than VASE in that it aims to address the problem with external calls. Also, solutions obtained from partial constraints may over-constrain the search space in symbolic execution.

## CHAPTER 8

### CONCLUSION AND OPEN PROBLEMS

As the most commonly used approach for assessing and improving software quality, testing has its inherent limitations. It is tedious and time-consuming to perform manually. Besides, test suites written by developers in house do not necessarily reflect how software is used by users in the real world. To address and mitigate these problems, and help improve the quality of in-house tests, I proposed an overall vision that contains both a comprehensive study that helps to understand the difference between in-house tests and field executions and a technique that attempts to improve the representativeness of in-house tests.

The first part of the dissertation presents the first study whose goal is to understand, quantify, and analyze similarities and differences between in-house testing and in-field usage. To do so, I have instrumented several software systems so that, when run in the field, they can collect various execution data. Then I used four models of software behavior — two based on coverage, one based on mutation analysis, and one based on temporal behavioral invariants. My results show that, for all the four models considered, there are gaps between how developers test, or how they expect users to use their software, and how users actually use this software in the field. Although still preliminary, I believe that these results are significant because they were obtained from analyzing field data collected from real systems used by real users or real developers (in the case of the libraries and their client projects).

After identifying the gap that exists between in-house tests and field executions and the test model to represent this gap, I present Replica, a technique capable of generating new tests that mimic field executions and that exercise behaviors not yet tested in-house but observed in the field. To do so, my technique collects lightweight execution data when the software is used in the field, identifies untested behavior (expressed as invariant violations), and uses an incremental, guided symbolic execution to generate test inputs that exercise

such behavior. I implemented my technique and evaluated it using field data collected from real applications and real users. These results, albeit still preliminary, show that Replica is able to generate tests that: (1) mimic the behavior of field executions in terms of invariant violations, and (2) reveal a large percentage of the seeded faults (mutants) that are revealed by field executions but not by in-house tests. The results also show that my technique outperforms a traditional symbolic execution approach that does not take advantage of data from the field.

To further improve the efficiency of test generation, I developed VASE, a technique leveraging observed execution data to reduce the cost of constraint solving. The technique collects variable values from concrete executions and infers properties from these values. The properties are then leveraged in symbolic execution to provide additional information for constraint solving. I evaluated the technique using real-world programs. The results confirmed that VASE is able to improve the performance of constraint solving and thus improve the effectiveness of test generating using symbolic execution.

There are several open problems that could be potentially addressed in order to make my techniques work in practice.

First, collecting field execution data and tracking concrete values impose overhead for user execution, which is not ideal. Therefore, one possible direction is to investigate more efficient data collection mechanisms to reduce the runtime overhead.

Second, like many other input generation techniques, my approaches currently do not generate oracles automatically. A possible extension to my dissertation is develop techniques that can generate or derive oracles for generated inputs.

Third, my VASE technique uses concrete values observed from real executions to reduce the cost of constraint solving. One variation of the technique is to explore the effectiveness of combining fuzzing techniques with symbolic execution. Fuzzing can produce large amount of random data efficiently and these data can be used to replace symbolic variables in order to reduce the cost of constraint solving.

## REFERENCES

- [1] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, “Synoptic: Studying logged behavior with inferred models,” in *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE)*, Szeged, Hungary, 9, 5, pp. 448–451.
- [2] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, “Unifying FSM-inference algorithms through declarative specification,” in *International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 24, 22, pp. 252–261.
- [3] ———, “Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms,” *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 4, pp. 408–428, 2015.
- [4] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, “Behavioral resource-aware model inference,” in *International Conference on Automated Software Engineering (ASE)*, Västerås, Sweden, 19, 15, pp. 19–30.
- [5] I. Krka, Y. Brun, and N. Medvidovic, “Automatic mining of specifications from invocation traces and method invariants,” in *International Symposium on the Foundations of Software Engineering (FSE)*, (FSE), Hong Kong, China, 22, 16, pp. 178–189.
- [6] G. J. Myers and C. Sandler, *The art of software testing*. USA: John Wiley & Sons, Inc., 2004, ISBN: 0471469122.
- [7] *Staged rollout*, <https://support.google.com/googleplay/android-developer/answer/6346149?hl=en>, 2019.
- [8] M. Young, “Perpetual testing,” Univ. of Oregon, Tech. Rep. Technical Report AFRL-IFRS-TR-2003-32, 2003.
- [9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *International Conference on Software Engineering (ICSE)*, Minneapolis, MN, USA, 2007, pp. 75–84.
- [10] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08, San Diego, California, 2008, pp. 209–224.

- [11] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE)*, Szeged, Hungary, 9, 5, pp. 448–451.
- [12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [13] A. W. Biermann and J. A. Feldman, “On the synthesis of finite-state machines from samples of their behavior,” *IEEE Transactions on Computers (TC)*, vol. 21, no. 6, pp. 592–597, 1972.
- [14] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehltz, and N. Rungta, “Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis,” *Automated Software Engineering*, vol. 20, pp. 391–425, 2013.
- [15] W. Visser, J. Geldenhuys, and M. 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*, ser. FSE ’12, Cary, North Carolina: ACM, 2012, 58:1–58:11, ISBN: 978-1-4503-1614-9.
- [16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.
- [17] K. N. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 685–718, Jun. 1991.
- [18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.
- [19] M. Gabel and Z. Su, “Javert: Fully automatic mining of general temporal properties from dynamic traces,” in *International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA, 2008.
- [20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, “Perracotta: Mining temporal API rules from imperfect traces,” in *International Conference on Software Engineering (ICSE)*, Shanghai, China, 2006, pp. 282–291, ISBN: 1-59593-375-1.
- [21] J. Yang and D. Evans, “Automatically inferring temporal properties for program evolution,” in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 340–351.

- [22] G. Jiang, H. Chen, and K. Yoshihira, “Efficient and scalable algorithms for inferring likely invariants in distributed systems,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, no. 11, pp. 1508–1523, 2007.
- [23] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.
- [24] R. Alur and T. A. Henzinger, “A really temporal logic,” *Journal of the ACM*, vol. 41, no. 1, pp. 181–203, 1994.
- [25] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE ’99, Los Angeles, California, USA, 1999, pp. 411–420, ISBN: 1-58113-074-0.
- [26] J. E. Cook and A. L. Wolf, “Discovering models of software processes from event-based data,” *ACM TOSEM*, vol. 7, no. 3, 1998.
- [27] D. Lo, L. Mariani, and M. Pezzè, “Automatic steering of behavioral model inference,” in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Amsterdam, The Netherlands, 2009, pp. 345–354, ISBN: 978-1-60558-001-2.
- [28] D. Lo and S.-C. Khoo, “QUARK: Empirical assessment of automaton-based specification miners,” in *Working Conference on Reverse Engineering (WCRE)*, 2006.
- [29] ———, “SMArTIC: Towards building an accurate, robust and scalable specification miner,” in *International Symposium on Foundations of Software Engineering (FSE)*, Portland, OR, USA, 2006, pp. 265–275, ISBN: 1-59593-468-5.
- [30] D. Lo and S. Maoz, “Scenario-based and value-based specification mining: Better together,” in *International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, 2010, pp. 387–396, ISBN: 978-1-4503-0116-9.
- [31] D. Lorenzoli, L. Mariani, and M. Pezzè, “Automatic generation of software behavioral models,” in *International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, pp. 501–510, ISBN: 978-1-60558-079-1.
- [32] S. P. Reiss and M. Renieris, “Encoding program executions,” in *International Conference on Software Engineering (ICSE)*, Toronto, ON, Canada, 2001, pp. 221–230, ISBN: 0-7695-1050-7.
- [33] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

- [34] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008, ISBN: 0521880386, 9780521880381.
- [35] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2003, pp. 141–154, ISBN: 1-58113-662-5.
- [36] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *Workshop on Eclipse Technology eXchange*, San Diego, California, 2005, pp. 35–39, ISBN: 1-59593-342-5.
- [37] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” In *International Workshop on Mining Software Repositories (MSR)*, Minneapolis, MN, USA, 2007, ISBN: 0-7695-2950-X.
- [38] R. C. Carrasco and J. Oncina, “Learning stochastic regular grammars by means of a state merging method,” in *International Colloquium on Grammatical Inference and Applications (ICGI)*, Alicante, Spain, 1994, pp. 139–152.
- [39] N. Walkinshaw and K. Bogdanov, “Inferring finite-state models with temporal constraints,” in *International Conference on Automated Software Engineering (ASE)*, L’Aquila, Italy, 2008, pp. 248–257, ISBN: 978-1-4244-2187-9.
- [40] P. G. Frankl and O. Iakounenko, “Further empirical studies of test effectiveness,” in *International Symposium on Foundations of Software Engineering (FSE)*, Lake Buena Vista, FL, USA, 1998, pp. 153–162, ISBN: 1-58113-108-9.
- [41] A. Gupta and P. Jalote, “An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 2, pp. 145–160, Feb. 2008.
- [42] A. Groce, “Coverage rewarded: Test input generation via adaptation-based programming,” in *International Conference on Automated Software Engineering (ASE)*, 2011, pp. 380–383, ISBN: 978-1-4577-1638-6.
- [43] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014, pp. 435–445, ISBN: 978-1-4503-2756-5.
- [44] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” In *International Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, China, 2014.

- [45] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” in *International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, 2010, pp. 147–158, ISBN: 978-1-60558-823-0.
- [46] W. Jin and A. Orso, “Bugredux: Reproducing field failures for in-house debugging,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12, Zurich, Switzerland, 2012, pp. 474–484, ISBN: 978-1-4673-1067-3.
- [47] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges,” in *International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015, pp. 201–211.
- [48] A. Arcuri, G. Fraser, and J. P. Galeotti, “Generating TCP/UDP network data for automated unit test generation,” in *10th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, 2015, pp. 155–165, ISBN: 978-1-4503-3675-8.
- [49] J. M. Rojas, G. Fraser, and A. Arcuri, “Automated unit test generation during software development: A controlled experiment and think-aloud observations,” in *International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, MD, USA, 2015, pp. 338–349.
- [50] A. A. Sawant and A. Bacchelli, “A dataset for api usage,” in *Working Conference on Mining Software Repositories (MSR)*, Florence, Italy, 2015, pp. 506–509.
- [51] V. Massol, T. O’Brien, and M. K. Loukides, *Maven: A developer’s notebook*, 1st ed. O’Reilly, 2005, ISBN: 0596007507.
- [52] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, 2014, pp. 437–440.
- [53] R. Just, F. Schweiggert, and G. M. Kapfhammer, “MAJOR: An efficient and extensible tool for mutation analysis in a java compiler,” in *International Conference on Automated Software Engineering (ASE)*, Lawrence, KS, USA, 2011, pp. 612–615, ISBN: 978-1-4577-1638-6.
- [54] *Execution comparison*, <http://www.cc.gatech.edu/~orso/software/ExecutionComparison/>, 2018.
- [55] *The LLVM Compiler Infrastructure*, <https://llvm.org/>, 2018.
- [56] *CURL*, <https://curl.haxx.se/>, 2018.

- [57] Y. Jia and M. Harman, “Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language,” in *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, ser. TAIC-PART ’08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 94–98, ISBN: 978-0-7695-3383-4.
- [58] Q. Wang, Y. Brun, and A. Orso, “Behavioral execution comparison: Are tests representative of field behavior?” In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 321–332.
- [59] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang, “Casper: An efficient approach to call trace collection,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16, St. Petersburg, FL, USA: ACM, 2016, pp. 678–690, ISBN: 978-1-4503-3549-2.
- [60] B. Kasikci, W. Cui, X. Ge, and B. Niu, “Lazy diagnosis of in-production concurrency bugs,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, Shanghai, China: ACM, 2017, pp. 582–598, ISBN: 978-1-4503-5085-3.
- [61] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for C,” in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, 2005, pp. 263–272, ISBN: 1-59593-014-0.
- [62] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’08/ETAPS’08, Budapest, Hungary, 2008, pp. 337–340, ISBN: 3-540-78799-2, 978-3-540-78799-0.
- [63] R. Dechter, *Constraint Processing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ISBN: 1558608907.
- [64] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 305–343, ISBN: 978-3-319-10575-8.
- [65] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV ’11)*, G. Gopalakrishnan and S. Qadeer, Eds., ser. Lecture Notes in Computer Science, Snowbird, Utah, vol. 6806, Springer, Jul. 2011, pp. 171–177.

- [66] V. Ganesh and D. L. Dill, “A decision procedure for bit-vectors and arrays,” in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, 2007, pp. 519–531.
- [67] B. Dutertre, “Yices 2.2,” in *Computer-Aided Verification (CAV’2014)*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 737–744.
- [68] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
- [69] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, “Gamma system: Continuous evolution of software after deployment,” in *International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, 2002, pp. 65–69, ISBN: 1-58113-562-9.
- [70] A. Orso, T. Apiwattanapong, and M. J. Harrold, “Leveraging field data for impact analysis and regression testing,” in *Proceedings of the 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Helsinki, Finland, 2003, pp. 128–137.
- [71] S. Elbaum and M. Diep, “Profiling deployed software: Assessing strategies and testing opportunities,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.
- [72] S. Elbaum and M. Hardojo, “An empirical study of profiling strategies for released software and their impact on testing activities,” in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, USA, 2004, pp. 65–75, ISBN: 1-58113-820-2.
- [73] M. Dyer, *The Cleanroom Approach to Quality Software Development*. New York, NY, USA: John Wiley & Sons, Inc., 1992, ISBN: 0-471-54823-5.
- [74] J. A. Whittaker and J. H. Poore, “Markov analysis of software specifications,” *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 1, pp. 93–106, Jan. 1993.
- [75] C. Pavlopoulou and M. Young, “Residual test coverage monitoring,” in *International Conference on Software Engineering (ICSE)*, 1999, pp. 277–284.
- [76] D. M. Hilbert and D. F. Redmiles, “Large-scale collection of usage data to inform design,” in *IFIP TC.13 International conference on human-computer interaction*, IOS Press, 2001, pp. 569–576.
- [77] S. Rao and A. Kak, “Retrieval from software libraries for bug localization: A comparative study of generic and composite text models,” in *Proceedings of the 8th*

*Working Conference on Mining Software Repositories (MSR)s*, Honolulu, HI, USA, 2011, pp. 43–52, ISBN: 978-1-4503-0574-7.

- [78] Q. Wang, C. Parnin, and A. Orso, “Evaluating the usefulness of ir-based fault localization techniques,” in *International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, MD, USA, 2015, pp. 1–11, ISBN: 978-1-4503-3620-8.
- [79] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, “Rebucket: A method for clustering duplicate crash reports based on call stack similarity,” in *International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, 2012, pp. 1084–1093, ISBN: 978-1-4673-1067-3.
- [80] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, “Crashlocator: Locating crashing faults based on crash stacks,” in *International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, 2014, pp. 204–214, ISBN: 978-1-4503-2645-2.
- [81] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, “Comparing non-adequate test suites using coverage criteria,” in *International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, 2013, pp. 302–313, ISBN: 978-1-4503-2159-4.
- [82] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” *SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 45–57, Oct. 2002.
- [83] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, 2001.
- [84] C. M. Antunes and A. L. Oliveira, “Temporal data mining: An overview,” in *Workshop on Temporal Data Mining*, San Francisco, CA, USA, 2001.
- [85] C. Marceau, “Characterizing the behavior of a program using multiple-length n-grams,” in *Workshop on New Security Paradigms (NSPW)*, Ballycotton, County Cork, Ireland, 2000, pp. 101–110, ISBN: 1-58113-260-3.
- [86] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, “Leveraging existing instrumentation to automatically infer invariant-constrained models,” in *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, 9, 5, pp. 267–277.
- [87] T. Ziadi, M. A. A. da Silva, L. M. Hillah, and M. Ziane, “A fully dynamic approach to the reverse engineering of UML sequence diagrams,” in *International Conference*

*on Engineering of Complex Computer Systems (ICECCS)*, Las Vegas, NV, USA, 2011, pp. 107–116, ISBN: 978-0-7695-4381-9.

- [88] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker, “Learning communicating automata from MSCs,” *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 3, pp. 390–408, 2010.
- [89] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, “Inferring models of concurrent systems from logs of their behavior with csight,” in *International Conference on Software Engineering (ICSE)*, Hyderabad, India, 6, 4, pp. 468–479.
- [90] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, “Mining message sequence graphs,” in *International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, 2011, pp. 91–100, ISBN: 978-1-4503-0445-0.
- [91] C. Csallner and Y. Smaragdakis, “Jcrasher: An automatic robustness tester for java,” *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.
- [92] M. Harman and P. McMinn, “A theoretical and empirical study of search-based testing: Local, global, and hybrid search,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar. 2010.
- [93] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005, pp. 213–223, ISBN: 1-59593-056-6.
- [94] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, “Testability transformation,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, Jan. 2004.
- [95] C. Cadar and D. Engler, “Execution generated test cases: How to make systems code crash itself,” in *International Conference on Model Checking Software (SPIN)*, San Francisco, CA, USA, 2005, pp. 2–23, ISBN: 3-540-28195-9, 978-3-540-28195-5.
- [96] J. Röβler, G. Fraser, A. Zeller, and A. Orso, “Isolating failure causes through test case generation,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, Minneapolis, MN, USA, 2012, pp. 309–319, ISBN: 978-1-4503-1454-1.
- [97] C. Zamfir and G. Candea, “Execution synthesis: A technique for automated software debugging,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10, Paris, France, 2010, pp. 321–334, ISBN: 978-1-60558-577-2.

- [98] N. Chen and S. Kim, “Star: Stack trace based automatic crash reproduction via symbolic execution,” *IEEE Transactions on Software Engineering*, vol. 41, no. 2, pp. 198–220, 2015.
- [99] M. Soltani, A. Panichella, and A. van Deursen, “A guided genetic algorithm for automated crash reproduction,” in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17, Buenos Aires, Argentina, 2017, pp. 209–220, ISBN: 978-1-5386-3868-2.
- [100] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, “Reproducing field failures for programs with complex grammar-based input,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 163–172.
- [101] X. Jia, C. Ghezzi, and S. Ying, “Enhancing reuse of constraint solutions to improve symbolic execution,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: ACM, 2015, pp. 177–187, ISBN: 978-1-4503-3620-8.
- [102] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, “Reusing constraint proofs in program analysis,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: ACM, 2015, pp. 305–315, ISBN: 978-1-4503-3620-8.
- [103] I. Erete and A. Orso, “Optimizing constraint solving to better support symbolic execution,” in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW ’11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 310–315, ISBN: 978-0-7695-4345-1.
- [104] C. S. Păsăreanu, N. Rungta, and W. Visser, “Symbolic execution with mixed concrete-symbolic solving,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA ’11, Toronto, Ontario, Canada, 2011, pp. 34–44, ISBN: 978-1-4503-0562-4.