

7-2017

Hybrid based approaches for software fault localization and specification mining

Tien-Duy B. LE
Singapore Management University

Follow this and additional works at: http://ink.library.smu.edu.sg/etd_coll_all

Part of the [Software Engineering Commons](#)

Citation

LE, Tien-Duy B.. Hybrid based approaches for software fault localization and specification mining. (2017). *Singapore Management University*. Dissertations and Theses Collection.

Available at: http://ink.library.smu.edu.sg/etd_coll_all/19

This PhD Dissertation is brought to you for free and open access by the Dissertations and Theses at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Dissertations and Theses Collection by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

HYBRID BASED APPROACHES FOR SOFTWARE FAULT
LOCALIZATION AND SPECIFICATION MINING

LE BUI TIEN DUY

SINGAPORE MANAGEMENT UNIVERSITY

2017

Hybrid Based Approaches for Software Fault Localization and Specification Mining

by

Tien-Duy B. Le

Submitted to School of Information Systems in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

David Lo (Supervisor/Chair)
Associate Professor of Information Systems
Singapore Management University

Lingxiao Jiang
Assistant Professor of Information Systems
Singapore Management University

Hady W. Lauw
Assistant Professor of Information Systems
Singapore Management University

Ivan Beschastnikh
Assistant Professor of Computer Science
University of British Columbia

SINGAPORE MANAGEMENT UNIVERSITY

2017

Copyright (2017) Le Bui Tien Duy

Hybrid Based Approaches for Software Fault Localization and Specification Mining

Tien-Duy B. Le

Abstract

Debugging programs and writing formal specifications are essential but expensive processes to maintain quality and reliability of software systems. Developers often have to debug and create specifications manually, which take a lot of their time and effort. Recently, several automated solutions have been proposed to help developers alleviate the cost of manual labor in the two processes. In particular, fault localization techniques help developer debug by accepting textual information in bug reports or program spectra (i.e., a record of which program elements are executed for each test case). Their output is a ranked list of program elements that are likely to be faulty. Developers then inspect the ranked list from beginning of the ranked list until root causes of the fault are found. On the other hand, many systems have no or lack of high quality formal specifications. To deal with the issue, researchers have proposed techniques to automatically infer specifications in a variety of formalism, such as finite state automaton (FSA). The inferred specifications can be used for many manual software processes, including debugging.

Unfortunately, to date, the efficacy of existing techniques in fault localization and specification mining are not perfect yet and more work is needed to employ the techniques to assist developers in debugging programs and writing formal specifications. In this dissertation, I propose a number of *hybrid based* approaches to improve the effectiveness of fault localization and specification mining. These hybrid based approaches combine the strength of different techniques and various sources of information to create more effective solutions. My goal is to lessen the

high expense of debugging and writing formal specifications in order to enhance the productivity of developers and software quality. To achieve that goal, I propose AML and Savant, which are new fault (bug) localization techniques, as well as SpecForge and DSM, which are new specification mining algorithms. Their contributions are summarized as follows

- **AML** is a new multi-modal technique that considers both bug reports and program spectra to localize bugs. The approach *adaptively* creates a bug-specific model to map a particular bug to its possible location, and introduces a novel idea of *suspicious words* that are highly associated to a bug.
- **Savant** is a new fault localization approach that employs a learning-to-rank strategy, using likely invariant *diffs* and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. Savant has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. At the end of these four steps, Savant produces a short ranked list of potentially buggy methods.
- **SpecForge** is a new specification mining approach that synergizes many existing specification miners. SpecForge decomposes FSAs that are inferred by existing miners into simple constraints, through a process we refer to as model fission. It then filters the outlier constraints and fuses the constraints back together into a single FSA (i.e., model fusion).
- **DSM** is a new approach that performs deep learning for mining FSA-based specifications. Our proposed approach uses test case generation to generate a rich set of execution traces for training a Recurrent Neural Network Based Language Model (RNNLM). From these execution traces, we construct a Prefix Tree Acceptor (PTA) and use the learned RNNLM to extract many

features. These features are subsequently utilized by clustering algorithms to merge similar automata states in PTA for constructing a number of FSAs. Then, our approach performs a model selection heuristic to estimate F-measure of FSAs and returns the one with highest estimated F-measure.

TABLE OF CONTENTS

	Page
List of Figures	iii
List of Tables	iv
Chapter 1: Introduction	1
1.1 Motivation	1
1.2 Dissertation Overview	4
Chapter 2: Literature Review	9
2.1 Software Fault Localization	9
2.2 Software Specification Mining	18
Chapter 3: Information Retrieval and Spectrum Based Bug Localization: Better Together	25
3.1 Introduction	25
3.2 Adaptive Multi-modal Bug Localization	28
3.3 Evaluation	37
3.4 Conclusion	47
Chapter 4: Learning-to-Rank Based Fault Localization using Likely In- variants	48
4.1 Introduction	48
4.2 Motivating Example	50
4.3 Savant	52
4.4 Evaluation	60
4.5 Conclusion	71
Chapter 5: Synergizing Specification Miners through Model Fissions and Fusions	73
5.1 Introduction	73
5.2 SpecForge	74
5.3 Evaluation	80

5.4	Conclusion	94
Chapter 6:	Deep Specification Mining	96
6.1	Introduction	96
6.2	Deep Specification Mining	99
6.3	Evaluation	109
6.4	Conclusion	118
Chapter 7:	Conclusion and Future Work	121
Bibliography	124

LIST OF FIGURES

Figure Number	Page
2.1 Bug Report 54460 of Apache Ant	16
3.1 AML’s Framework	29
3.2 AML vs. Baselines in terms of Top N ($N \in \{1, 5, 10\}$) and MAP. .	40
3.3 Contributions of AML Components in Bar Charts.	43
3.4 Integrator vs. SVM ^{rank} in Bar Charts.	44
4.1 Bug Report & Developer Patch for Bug 383 of Closure Compiler .	51
4.2 Overview of <i>Savant</i> ’s Architecture	52
4.3 <i>Savant</i> : Input Data Format for Model Learning	60
4.4 Top N ($N \in \{1, 5, 10\}$): <i>Savant</i> vs. Baselines in Bar Charts. . . .	69
4.5 MAP: <i>Savant</i> vs. Baselines in Bar Charts.	70
5.1 SpecForge Overview	75
5.2 FSA for “ <i>a</i> is never immediately followed by <i>a</i> ”.	80
5.3 Translations of LTL expressions to FSAs	81
5.4 Average Precision, Recall, and F-measure: SpecForge vs. Baselines.	87
5.5 Average Precision, Recall, and F-measure: SpecForge with Differ- ent Constraint Templates.	91
5.6 Average Precision, Recall, and F-measure: SpecForge with Differ- ent Constraint Selection Heuristics.	92
6.1 DSM’s overall framework.	99
6.2 Example of an unrolled Recurrent Neural Network	102
6.3 An example Prefix Tree Acceptor (PTA).	106
6.4 Average Precision, Recall, and F-measure: DSM vs. Baselines. . .	115

LIST OF TABLES

Table Number	Page
2.1 List of Spectrum-based Fault Localization (SBFL) Raw Statistics	10
2.2 Description of SBFL Raw Statistics	10
3.1 AML’s Dataset Description	37
3.2 Top N: AML vs. Baselines. $N \in \{1, 5, 10\}$	40
3.3 Mean Average Precision: AML vs. Baselines.	40
3.4 Contributions of AML Components	42
3.5 AML’s Integrator vs. SVM ^{rank}	43
3.6 Running Time of AML	45
3.7 Effect of Varying Number of Neighbors on AML	45
4.1 <i>Savant</i> ’s Dataset	61
4.2 <i>Savant</i> ’s Effectiveness	65
4.3 <i>Savant</i> : Effectiveness of Baseline Approaches (Part I)	66
4.4 <i>Savant</i> : Effectiveness of Baseline Approaches (Part II)	67
4.5 <i>Savant</i> : Effectiveness of Baseline Approaches (Part III)	68
4.6 <i>Savant</i> ’s Effectiveness Using Different Features	68
4.7 <i>Savant</i> : Varying Training Data Size	70
4.8 Running Time of <i>Savant</i>	71
5.1 SpecForge: List of Target Library Classes and Analyzed Programs	80
5.2 Precision, Recall, and F-measure: SpecForge with Default Configuration.	85
5.3 SpecForge: Effectiveness of Traditional 1-tail and Traditional 2-tail	87
5.4 SpecForge: Effectiveness of CONTRACTOR++	88
5.5 SpecForge: Effectiveness of of SEKT 1-tail and SEKT 2-tail	89
5.6 Precision, Recall, and F-measure: Optimistic TEMI and Pessimistic TEMI	90
5.7 Average Precision, Recall, and F-measure: SpecForge with Different Constraint Templates.	91
5.8 Average Precision, Recall, and F-measure: SpecForge with Different Constraint Selection Heuristics.	92
6.1 Extracted Features for An Automata State	107

6.2	Target Library Classes	109
6.3	DSM: Precision, Recall, and F-measure	113
6.4	DSM: Effectiveness of Traditional 1-tail and Traditional 2-tail	114
6.5	DSM: Effectiveness of SEKT 1-tail and SEKT 2-tail	116
6.6	DSM: Effectiveness of CONTRACTOR++ and Optimistic TEMI	117
6.7	DSM with standard RNN (DSM ^{RNN})	118
6.8	DSM with GRU (DSM ^{GRU})	119

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor Prof. David Lo for his generous advices with his great passion for research. Prof. Lo's guidance helped me in gaining research skills and experiences.

Next, I would like to say thank you to Prof. Ivan Beschastnikh, Prof. Claire Le Goues, Ferdian Thung, Le Dinh Xuan Bach, Tian Yuan, Pavneet S. Kocchar, members of Software Analytics Research (SOAR) group, and all co-authors for your supports and wonderful discussions during our collaborated works.

Besides, I would like to thank my dissertation committee: Prof. Lingxiao Jiang, Prof. Hady W. Lauw, and Prof. Ivan Beschastnikh for the comments and feedbacks on my dissertation.

Finally, I would like to thank my family for supporting and encouraging me throughout my PhD studies.

Chapter 1

INTRODUCTION

1.1 Motivation

Software developers often handle many manual tasks which are costly and take long-time to complete. Among these tasks, debugging programs and writing formal specification are the essential ones. Software systems are usually plagued with bugs that compromise system reliability, usability, and security. Developers have to debug to find and repair those bugs that make their programs behave unexpectedly. In fact, debugging is time consuming and expensive, which can contribute up to 80% of the total software cost for some projects [106]. On the other hand, software systems and libraries that are released without any documented specifications due to short time-to-market and rapid evolution of software. The unavailability of specifications makes code comprehension more difficult for developers, and software becomes more error-prone as bugs are introduced due to mistaken assumptions. However, constructing formal specifications requires developers to have requisite skills and motivation, as this takes significant time and manual effort [43]. Thus, there is a pressing need for automated solutions that improve productivity of developers when debugging programs and writing formal specifications. Recently, several fault localization and specification mining techniques have been proposed to reduce the expense of these manual processes, respectively, in many scenarios:

- **Fault Localization.** To help developers debug, fault (or bug) localization¹ techniques analyze the symptoms of a bug, and produce a list of pro-

¹Traditionally, fault localization and bug localization are two distinct families of debugging techniques. In a nutshell, faults are defects (i.e., incorrect program code) that cause failures

gram elements ranked based on their likelihood to contain the bug. These symptoms could be in the form of a description of a bug experienced by a user, or a failing test case. Existing bug localization techniques can be divided into two families: information retrieval (IR)-based bug localization techniques [95, 104, 127, 101], and spectrum-based bug localization techniques [40, 6, 96, 122, 124, 20, 58, 57, 60]. IR-based bug localization techniques typically analyze textual descriptions contained in bug reports and identifier names and comments in source code files. Spectrum-based bug localization techniques typically analyze program spectra that corresponds to program elements that are executed by failing and successful execution traces. In this dissertation, I focus on improving the effectiveness of spectrum-based fault (bug) localization by leveraging additional information from textual bug reports and likely invariants.

- **Specification Mining.** Due to the short time-to-market and rapid evolution of software, software systems and libraries are often released without any documented specifications. Even when a system includes formal specifications, these specifications may become quickly out of date as the software evolves [126]. Finally, developers often lack the necessary skill and motivation to write formal specifications, as this takes significant time and manual effort [43]. The unavailability of specifications negatively impacts the maintainability and reliability of systems. Without specifications developers find code comprehension more difficult, and software becomes more error-prone as bugs are introduced due to mistaken assumptions. Moreover, without a formal specification, developers cannot take advantage of some state-of-

in software applications. On the other hand, bugs are faults reported by software testers or users, and logged onto a bug tracking system. Usually, a bug accompanies with a textual bug report describing its syndrome, but a fault often does not have. Even though fault localization and bug localization techniques are applied in different settings and inputs, intuitions behind fault localization techniques can be utilized to localize bugs, and vice versa. Therefore, in this dissertation I use “fault localization” and “bug localization” interchangeably.

the-art bug finding and testing tools that require formal specifications as an input [19, 78]. To address the unavailability of formal specifications, Krka et al. [46], Beschastnikh et al. [15, 14], Lo et al. [26, 66], Mariani et al. [77], and many others have proposed finite state automaton (FSA)-based specification mining algorithms. In this dissertation, I focus on techniques that extract a model in the form of a finite state automaton (FSA) by analyzing the execution traces of systems or libraries of interest.

The above mentioned fault localization approaches [95, 104, 127, 101, 40, 6, 96, 122, 124, 20, 58, 57, 60] can be further improved to localize more bugs more accurately. In particular, most of fault localization approaches only consider one kind of symptom or one source of information, i.e., only bug reports or only execution traces. This is a limiting factor since hints of the location of a bug may be spread in both bug report and execution traces; and some hints may only appear in one but not the other. Furthermore, fault localization approaches can utilize the advantage of formal specifications (e.g., likely invariants [30] etc.) to detect faulty program elements that violate properties regulated by formal specifications.

Recently, Krka et al. [46] combine both of execution traces and likely invariants to mine specifications. However, specification miners (i.e., mining approaches) have not been combined together to become a single one before. Recently, deep learning (also known as deep machine learning) methods are proposed to learn representations of data with multiple levels of abstraction [52]. Researchers have utilized deep learning in various domains, including solving challenging tasks in software engineering such as defect prediction [112], code clone detection [113], etc. Nevertheless, deep learning methods have not been employed for mining specifications before. In fact, the power of deep learning methods to learn complex representations of execution traces by leveraging deep neural networks (i.e., networks with many layers) can be used to boost the effectiveness of existing state-

of-the-art specification miners. Therefore, similar to fault localization, I believe there are many opportunities to further improve existing mining approaches and bring them closer to adoption in practice.

1.2 *Dissertation Overview*

In this dissertation, I propose a number of hybrid based approaches for fault localization and specification mining. In terms of design paradigm, the core idea of hybrid based methodologies is to construct an effective solution by combining the strengths of different methods and utilizing various sources of informations. For instance, it is possible for fault localization techniques to process both textual bug reports and execution traces. It is also potential to combine many specification miners to create a more powerful one. In fact, existing state-of-the-art automated solutions in fault localization and specification mining have not fully utilized the power of hybrid based paradigm.

Hybrid based Fault Localization: Most of existing fault localization techniques only process execution traces of passed and failed test cases, or textual bug reports separately. Nevertheless, existing hybrid based fault localization approaches could be better if they can blend various sources of information (i.e., textual bug reports, execution traces, likely invariants, etc.) from input faults or bugs into one single framework. Therefore, I propose the following hybrid based approaches to bring fault localization closer to adoption in practice:

1. **AML** (Adaptive Multi-modal bug Localization): a multi-modal technique that considers both bug reports and program spectra to localize bugs. AML adaptively creates a bug-specific model to map a particular bug to its possible location, and introduces a novel idea of suspicious words that are highly associated to a bug. In the nutshell, my approach has three components: AML^{Text} , AML^{Spectra} , and AML^{SuspWord} . AML^{Text} only processes the textual description in bug reports, and AML^{Spectra} only handles program spectra.

On the other hand, AML^{SuspWord} considers suspicious words learned by analyzing textual description and program spectra together. AML^{SuspWord} calculates the *suspicious scores* of words that appear in comments or identifiers of various program elements. The approach associates a program element to a set of words and the suspiciousness of a word can then be estimated based on the number of times the corresponding program elements appear in failing or correct execution traces. Each of these components would output a score for each program element, and AML computes the weighted sum of these scores. The final score is *adaptively* computed for each individual bug by tuning these weights. To evaluate AML, I apply the approach on 157 real bugs from four software systems, and compare its effectiveness with a state-of-the-art IR-based bug localization method (i.e., LR [120]), a state-of-the-art spectrum-based bug localization method (i.e., Multric [118]), and three state-of-the-art multi-modal feature location methods (i.e., PROMESIR [92], DIT^A and DIT^B [28]) that are adapted for bug localization. Experiments show that AML can outperform the baselines by at least 47.62%, 31.48%, 27.78%, and 28.80% in terms of number of bugs successfully localized when a developer inspects 1, 5, and 10 program elements (i.e., Top 1, Top 5, and Top 10), and Mean Average Precision (MAP) respectively.

2. **Savant**: a new fault localization approach that employs a learning-to-rank strategy, using likely invariant diffs and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. Savant has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. At first, the approach selects a subset of test cases, including both the failing tests and only those passing tests that cover similar program elements. Next, it uses Daikon [30] to learn likely invariants of those methods that are executed by the failing executions. By *diff*-ing various sets of invariants inferred from passed

and failed execution traces, we identify suspicious methods where invariants inferred from one set of executions do not hold in another. Next, we convert the invariant diffs into a set of features. We also use the suspiciousness scores computed by several SBFL formulae for the suspicious methods as features. All of the extracted features are then provided as input to a learning-to-rank algorithm. The learning-to-rank algorithm learns a ranking model based on a training set of fixed bugs which differentiates invariant differences of faulty from non-faulty methods. At the end of these four steps, the approach produces a short ranked list of potentially buggy methods. To evaluate Savant, I apply the approach on 357 real-life bugs from 5 programs from the Defects4J benchmark. On average, Savant can identify the correct buggy method for 63.03, 101.72, and 122 bugs at the top 1, 3, and 5 positions in the produced ranked lists. I also compared Savant against several state-of-the-art spectrum-based fault localization baselines (i.e., theoretically best fault localization formulas [116], genetic programming generated fault localization formulas [121], and Multric [118]). According to the results, Savant can successfully locate 57.73%, 56.69%, and 43.13% more bugs at top 1, top 3, and top 5 positions than the best performing baseline, respectively.

Hybrid based Specification Mining: Recently, Krka et al. [46] have a number of state-of-the-art approaches that put together information of execution traces and likely invariants to mine FSAs. Despite recent achievements in quality of inferred specifications, more work is needed to further improve quality of mined specifications. Existing specification could be better if they are able to combine different specification miners together or leverage advanced machine learning technologies for mining specifications. Therefore, I propose the following specification mining approaches to improve quality of inferred specifications:

1. **SpecForge:** a new specification mining approach that synergizes many existing specification miners. SpecForge first uses existing specification miners

to infer a set of FSAs. It then uses these to generate a superior FSA. The approach first performs *model fissions* to extract important constraints that are common across the mined FSAs. SpecForge then performs *model fusions* to combine the extracted constraints into one FSA model. Both model fission and model fusion processes are completely automated. In this work, we use a set of 6 constraint templates to generate constraints, some of which were proposed by Dwyer et al. [29] and Beschastnikh et al. [14]. SpecForge checks whether one or more instances of these constraint templates are observed in a mined model. Constraints corresponding to models generated by various specification miners are then merged together while the outlier constraints are identified and omitted. To evaluate SpecForge, I apply the approach on execution traces of 10 programs, which includes 5 programs from DaCapo benchmark, to infer behavioral models of 13 library classes. Our results show that SpecForge achieves an average precision, recall and F-measure of 90.57%, 54.58%, and 64.21% respectively. SpecForge outperforms the best performing baseline proposed by Krka et al. [46] by 13.75% in terms of F-measure.

2. **DSM** (Deep Specification Miner): a hybrid based approach that performs deep learning for mining FSA-based specifications. DSM takes as input a target library class C and employs an automated test case generation tool to generate thousands of test cases. The goal of this test case generation process is to capture a rich set of valid sequences of invoked methods of C . Next, the approach performs deep learning on execution traces of generated test cases to train a Recurrent Neural Network Language Model (RNNLM) [80]. After this step, DSM constructs a Prefix Tree Acceptor (PTA) from the execution traces and leverages the learned language model to extract a number of interesting features from PTA’s nodes. These features are then input to clustering algorithms for merging similar states (i.e., PTA’s nodes). The

output of an application of a clustering algorithm is a simpler and more generalized FSA that reflects the training execution traces. Finally, our approach predicts the accuracy of constructed FSAs (generated by different clustering algorithms considering different settings) and outputs the one with highest predicted value of F-measure. To evaluate DSM, I perform the approach to mine specifications of 11 target library classes. The empirical analysis shows that DSM achieves an average Precision, Recall, and F-measure of 82.76%, 72.3%, and 71.97%, respectively. In comparison with specification mining algorithms proposed by Krka et al. [46] and SpecForge, DSM is more effective than the best baseline by 28.22% in terms of average F-measure.

The structure of the remainder of the dissertation is as follows. Chapter 2 highlights recent research studies in fault localization and specification mining. Next, Chapters 3, 4, 5, and 6 describe details of AML, Savant, SpecForge, and DSM, respectively. Chapter 7 concludes this dissertation and mentions future work.

Chapter 2

LITERATURE REVIEW

This chapter briefly describes research studies that are related to my works. Section 2.1 highlights existing approaches in spectrum-based and information retrieval based fault localization. Section 2.2 discusses recent works in rule based and automaton based specification mining.

2.1 Software Fault Localization

2.1.1 Spectrum-Based Fault Localization

Spectrum-based fault localization (SBFL) takes as input a faulty program and two sets of test cases. One is a set of failed test cases, and the other one is a set of passed test cases. SBFL then instruments the target program, and records program spectra that are collected when the set of failed and passed test cases are run on the instrumented program. Each of the collected program spectrum contains information of program elements that are executed by a test case. Various tools can be used to collect program spectra as a set of test cases are run.

Based on these spectra, SBFL typically computes some raw statistics for every program element. Tables 2.1 and 2.2 summarize some raw statistics that can be computed for a program element e . These statistics are the counts of unsuccessful (i.e., failed), and successful (i.e., passed) test cases that execute or do not execute e . If a successful test case executes program element e , then we increase $n_s(e)$ by one unit. Similarly, if an unsuccessful test case executes program element e , then we increase $n_f(e)$ by one unit. SBFL uses these statistics to calculate the suspiciousness scores of each program element. The higher the suspiciousness score, the more likely the corresponding program element is the faulty element.

Table 2.1: Raw Statistics for Program Element e

	e is <i>executed</i>	e is <i>not executed</i>
unsuccessful test	$n_f(e)$	$n_f(\bar{e})$
successful test	$n_s(e)$	$n_s(\bar{e})$

Table 2.2: Raw Statistic Description

Notation	Description
$n_f(e)$	Number of unsuccessful test cases that execute program element e
$n_f(\bar{e})$	Number of unsuccessful test cases that do not execute program element e
$n_s(e)$	Number of successful test cases that execute program element e
$n_s(\bar{e})$	Number of successful test cases that do not execute program element e
n_f	Total number of unsuccessful test cases
n_s	Total number of successful test cases

After the suspiciousness scores of all program elements are computed, program elements are then sorted in descending order of their suspiciousness scores, and sent to developers for manual inspection.

There are a number of SBFL techniques which propose various formulas to calculate suspiciousness scores. Among these techniques, Tarantula is a popular one [40]. Using the notation in Table 2.2, the following is the formula that Tarantula uses to compute the suspiciousness score of program element e :

$$Tarantula(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_f(e)}{n_f} + \frac{n_s(e)}{n_s}}$$

The main idea of Tarantula is that program elements that are executed by failed test cases are more likely to be faulty than the ones that are not executed by failed test cases. Thus, Tarantula assigns a non-zero score to program element e that

has $n_f(e) > 0$. In addition to Tarantula, Ochiai [7] is also a well-known formula:

$$Ochiai(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_s(e))}}$$

Recently, state-of-the-art SBFL approaches that have been recently demonstrated as either theoretically more optimal or high-performing than the other ones [116, 121, 118]. Xie *et al.* [116] theoretically analyze the manually-created SBFL formulas used in previous studies to compute suspiciousness scores and demonstrate that *ER1* and *ER5* are the two best SBFL families. They are computed as follows:

$$ER1^a(e) = \begin{cases} -1, & \text{if } n_f(e) < n_f \\ n_s - n_s(e), & \text{if } n_f(e) = n_f \end{cases}$$

$$ER1^b(e) = n_f(e) - \frac{n_s(e)}{n_s(e) + n_s(\bar{e}) + 1}$$

$$ER5^a(e) = n_f(e)$$

$$ER5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_s(e) + n_s(\bar{e})}$$

$$ER5^c(e) = \begin{cases} 0, & \text{if } n_f(e) < n_f \\ 1, & \text{if } n_f(e) = n_f \end{cases}$$

Xie *et al.* [117] further analyze SBFL formulas generated by running an automatic genetic programming (GP) algorithm [121]. The best GP-generated SBFL formulas are:

$$GP02(e) = 2 \times (n_f(e) + \sqrt{n_s}) + \sqrt{n_s(e)}$$

$$GP03(e) = \sqrt{|n_f(e)^2 - \sqrt{n_s(e)}|}$$

$$GP13(e) = n_f(e) \times \left(1 + \frac{1}{2 \times n_s(e) + n_f(e)}\right)$$

$$GP19(e) = n_f(e) \times \sqrt{|n_s(e) - n_f(e) + n_f - n_s|}$$

Xuan and Monperrus propose Multric, a compositional SBFL technique that uses a learning-to-rank algorithm [118] to combine the results of 25 previously-proposed SBFL formulas into a model that produces a single ranked list of potentially-buggy program elements. The 25 formulas are used to compute suspiciousness scores of program elements as usual; these scores are then treated as features input to a learning-to-rank algorithm.

Pytlik *et al.* propose Carrot, a SBFL technique that also leverages likely invariants [94]. Carrot mines a set of likely invariants [30] from executions of successful test cases, and observes how the invariants change when executions of failed test cases are incorporated. These differences indicate potential bug locations. Carrot considers the following six invariant types: equality, sum, less than, constant equality, value sets, and pairs of value sets.

Sahoo et al. [102] extend Pytlik et al.'s approach by adding test case generation and backward slicing to reduce the number of program elements to inspect, and modify parts of failing test cases to create new successful tests. Otherwise, it requires specifications that describe the test cases. Sahoo et al. evaluated their approach by localizing 6 real faults in MySQL, 1 real fault in Squid, and 1 real fault in Apache 2.2.

Pearson et al. [89] evaluate the effectiveness of many spectrum-based fault localization techniques, including Ochiai [7] and Tarantula [40], on 2995 artificial faults and 310 real faults. The study discovers that results of analyzed fault localization on artificial faults are inconsistent with the ones on real faults i.e., 40% of results on artificial faults are reversed and the other 60% of the results are statistically insignificant. In this dissertation, I assess the effectiveness of the proposed fault (bug) localization approaches (i.e., AML and SpecForge) using real bugs collected from repositories of real software systems (e.g., Apache Ant, AspectJ, Rhino etc.).

AML - Key Differences. All of mentioned above approaches only consider a

program spectra which is a record of program elements that are executed in failed and successful executions, and generate a ranked list of program elements. This is a limiting factor since hints of the location of a bug or a fault may be spread in both bug report and execution traces; and some hints may only appear in one but not the other. AML (see Chapter 3) is proposed to address this limitation of existing spectrum-based fault (bug) localization by considering both bug reports and execution traces. We refer to AML as a multimodal fault (bug) localization approach since we need to consider multiple modes of inputs (i.e., bug reports and program spectra). Furthermore, AML is the first to compute suspicious words and use these words to help bug localization. Previous studies in spectrum-based fault localization only compute suspiciousness scores of program elements.

Savant - Key Differences. Compared to Tarantula [40], Ochiai [7], theoretically best SBFL formulas [116], and GP generated SBFL formulas [121], Savant (see Chapter 4) is different as the approach uses a learning-to-rank machine learning approach to sort program elements by analyzing both classic suspiciousness scores and inferred likely invariants observed on passing and failing test cases.

Compared to Multric [118], Savant, also uses a learning-to-rank algorithm and includes suspiciousness scores as features, but we include a substantively different set of features extracted from invariant differences. Importantly, Savant localizes root causes of bugs in suspicious methods where invariant differences occur, rather than in all methods, as Multric does.

Compared to Pytlik et al.’s approach (i.e., Carrot), Savant uses a much larger set of invariant types (that is, all invariants produced by Daikon [30], rather than the six considered by Carrot), employ a method clustering and test case selection heuristic for performance, and use invariants and suspiciousness scores as features to rank program elements.

Compared to Sahoo et al.’s approach [102] Savant differs from theirs in several respects. First, instead of *filtering* invariants, Savant uses them as features to

rank program elements. It is possible to combine the filtering with the ranking approach in future work. Second, Savant avoids several restrictions and limitations of the previous work, in that it is not limited to programs with test oracles, programs where character-level rewriting of test cases makes sense, nor programs with test case specifications. Finally, our evaluation is substantially larger. We unfortunately cannot evaluate Sahoo et al.’s approach on our dataset since we do lack test oracles (unless we manually create them for all failing test cases), character-level deletion of test cases do not make sense for most of the programs (except for some bugs from Closure Compiler), and the test cases do not come with specifications. Moreover, Sahoo et al.’s approach is demonstrated on C rather than Java programs.

2.1.2 Information Retrieval Based Fault Localization

IR-based bug localization techniques consider an input bug report (i.e., the text in the summary and description of the bug report – see Figure 4.1) as a query, and program elements in a code base as documents, and employ information retrieval techniques to sort the program elements based on their relevance with the query. The intuition behind these techniques is that program elements sharing many common words with the input bug report are likely to be relevant to the bug. By using text retrieval models, IR-based bug localization computes the similarities between various program elements and the input bug report. Then, program elements are sorted in descending order of their textual similarities to the bug report, and sent to developers for manual inspection.

All IR-based bug localization techniques need to extract textual contents from source code files and preprocess textual contents (either from bug reports or source code files). First, comments and identifier names are extracted from source code files. These can be extracted by employing a simple parser. Next, after the textual contents from source code and bug reports are obtained, they have to be

preprocessed. The purpose of text preprocessing is to standardize words in source code and bug reports. There are three main steps: text normalization, stopword removal, and stemming:

- Text normalization breaks an identifier into its constituent words (tokens), following camel casing convention. Following the work by Saha et al. [101], the original identifier names are retained in preprocessed text.
- Stopword removal removes punctuation marks, special symbols, number literals, and common English stopwords [4]. It also removes programming keywords such as *if*, *for*, *while*, etc., as these words appear too frequently to be useful enough to differentiate between documents.
- Stemming simplifies English words into their root forms. For example, "processed", "processing", and "processes" are all simplified to "process". This increases the chance of a query and a document to share some common words.

There are many IR techniques that have been employed for bug localization. We highlight a popular IR technique namely *Vector Space Model* (VSM). In VSM, queries and documents are represented as vectors of weights, where each weight corresponds to a term. The value of each weight is usually the *term frequency—inverse document frequency* (TF-IDF) of the corresponding word. Term frequency refers to the number of times a word appears in a document. Inverse document frequency refers to the number of documents in a corpus (i.e., a collection of documents) that contain the word. The higher the term frequency and inverse document frequency of a word, the more important the word would be. In this work, given a document d and a corpus C , the TF-IDF weight of a word w

Bug 54460

Summary: Base64Converter not properly handling bytes with MSB set (not masking byte to int conversion)

Description: Every 3rd byte taken for conversion (least significant in triplet is not being masked with added to integer, if the msb is set this leads to a signed extension which overwrites the previous two bytes with all ones ...

Figure 2.1: Bug Report 54460 of Apache Ant

is computed as follows:

$$\text{TF-IDF}(w, d, C) = \log(f(w, d) + 1) \times \log \frac{|C|}{|d_i \in C : w \in d_i|}$$

where $f(w, d)$ is the number of times word w appears in document d .

After computing a vector of weights for the query and each document in the corpus, we calculate the cosine similarity of the query's vector and the document's vector. The cosine similarity between query q and document d is given by:

$$\text{sim}(q, d) = \frac{\sum_{w \in (q \cap d)} \text{weight}(w, q) \times \text{weight}(w, d)}{\sqrt{\sum_{w \in q} \text{weight}(w, q)^2} \times \sqrt{\sum_{w \in d} \text{weight}(w, d)^2}}$$

where $w \in (q \cap d)$ means word w appears both in the query q and document d . Also, $\text{weight}(w, q)$ refers to the weight of word w in the query q 's vector. Similarly, $\text{weight}(w, d)$ refers to the weight of word w in the document d 's vector.

Various IR-based fault localization approaches that employ information retrieval techniques to calculate the similarity between a bug report and a program element (e.g., a method or a source code file) have been proposed [95, 73, 104, 127, 101, 110, 111, 120]. Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization [73]. Then, Rao and Kak evaluated the utility of many standard IR techniques for bug localization including VSM and

Smoothed Unigram Model (SUM) [95]. In the IR community, historically, VSM is proposed very early (four decades ago by Salton et al. [103]), followed by many other IR techniques, including SUM and LDA, which address the limitations of VSM.

Recently, a number of approaches which considers information aside from text in bug reports to better locate bugs were proposed. Sisman and Kak proposed a version history aware bug localization technique which considers past buggy files to predict the likelihood of a file to be buggy and uses this likelihood along with VSM to localize bugs [104]. Around the same time, Zhou et al. proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs [127]. Next, Saha et al. proposed an approach that takes into account the structure of source code files and bug reports and employs structured retrieval for bug localization, and it performs better than BugLocator [101]. Subsequently, Wang and Lo proposed an approach that integrates the approaches by Sisman and Kak, Zhou et al. and Saha et al. for more effective bug localization [110]. Most recently, Ye et al. proposed an approach named LR that combines multiple ranking features using learning-to-rank to localize bugs, and these features include surface lexical similarity, API-enriched lexical similarity, collaborative filtering, class name similarity, bug fix recency, and bug fix frequency [120].

AML and Savant - Key Differences. When a developer receives a bug report, he/she first needs to replicate the error described in the report. In this process, he/she is creating at least one failing test case whose execution traces can be investigated to locate faulty program elements. All mentioned above IR-based fault localization approaches are limited to only analyze textual descriptions contained in bug reports and identifier names and comments in source code files. This is a limiting factor since hints of the location of a bug may be spread in both bug

report and execution traces; and some hints may only appear in one but not the other. AML (see Chapter 3) is proposed to address this limitation by processing both bug reports and test cases. AML also computes suspicious words and use these words to help bug localization. Previous studies in IR-based fault localization only compute suspiciousness scores of program elements. On the other hand, Savant (see Chapter 4) is a spectrum-based fault localization approach that only analyzes execution traces of unsuccessful and successful test cases.

2.2 Software Specification Mining

Formal specifications are useful for describing and understanding behaviors of software systems. Especially, specifications are helpful when developers try to find bugs or understand their programs. However, writing formal specifications takes significant time and manual effort. Developers are required to have the necessary skill and motivation to complete the task. In fact, software systems are often developed and released without formal specifications. In order to support developers, software engineering research community has proposed several approaches that automatically mine specifications from various sources of information (i.e., execution traces or source code) and in various forms (i.e., temporal rules or finite state automata). These approaches are divided into two major families: static and dynamic specification mining. Static approaches construct specifications from implementations (i.e., source code) of software systems, while dynamic approaches infer specifications by observing executions of software systems. In this dissertation, I focus on dynamic approaches that analyze execution traces to mine specifications, and refer to “dynamic specification mining” as “specification mining” for short.

Dynamic specification mining approaches can be further categorized to automaton-based and rule-based approaches. The former category’s miners infer final specifications in the form of finite state automata (see Section 2.2.1), while the later category’s miners output final specifications in the form of tem-

poral rules or properties (see Section 2.2.2). A specification miner should have generalization capability. Specifically, each specification miner has its own way to generalize behaviors of a software system encoded in input execution traces in order to infer specifications that reflect unobserved behaviors of the given system [13]. Generalization depends on features of analyzed systems, environments, forms and usages of mined specifications [13]. Furthermore, generalization is particularly good for specification miners to capture infrequent behaviors that are rarely observed during executions of software systems. However, generalization is not always correct as inferred unobserved behaviors possibly reflect infeasible executions of systems that never take place in practice. When input execution traces increasingly capture all or most of properties of a software system, generalization is less likely needed as most of behaviors are likely to be covered in the traces.

2.2.1 Automaton-based Specification Mining.

Krka et al. propose several specification mining algorithms that can infer a finite state automaton (FSA) from execution traces by leveraging value-based invariants that are inferred by Daikon [46]. They propose a number of algorithms namely CONTRACTOR++, state-enhanced k-tails (SEKT), and trace-enhanced MTS inference (TEMI).

- **k-tails:** k-tails is a classic algorithm proposed by Biermann and Feldman [16] to infer a FSA from execution traces. The algorithm takes as input a set of execution traces and a parameter k . To infer a FSA that describes the input execution traces, *k-tails* first builds a prefix tree acceptor (PTA) that accepts all of the input traces. A PTA is an automaton in the form of a tree, where every common prefix among the input traces corresponds to one state. Next, *k-tails* merges every two states of the PTA that have identical sequences of the next k method invocations (i.e., k-tails). The effectiveness of the *k-tails* algorithm depends the choice of k and the quality of its in-

put traces. If the value of k is small, k -tails might lead to incorrect state merges. If the value of k is large, then there are fewer merges, which limits the generalization of the inferred specifications. Similarly, if the number of input traces is small, then the inferred FSA might not accurately capture the correct specifications. In this work, we are interested in two variants of k -tails with the value of $k \in \{1, 2\}$. We refer to these two variants as *traditional 1-tails* and *traditional 2-tails*.

- **CONTRACTOR++:** CONTRACTOR++ is a recently proposed algorithm by Krka et al. [46] that uses inferred value-based program invariants to aid the construction of a FSA from execution traces. CONTRACTOR++ first runs Daikon [30] to infer several families of value-based program invariants; these include relational invariants (e.g., $x > 5$), null invariants (e.g., x is null), and size invariants (e.g., $x.size() > 5$). It then calls CONTRACTOR [27] which is able to construct a FSA from a set of invariants by running SMT solvers. CONTRACTOR characterizes each state in the constructed FSA by a set of methods that are enabled on that state. A legal state is a state in which the preconditions of the enabled methods are consistent with one another. CONTRACTOR creates a transition for a method from a source state to a target state if that method has both its precondition satisfied in the source state as well as its postcondition satisfied in the target state.
- **SEKT:** State-enhanced k-tails (SEKT) is another recently proposed algorithm by Krka et al. [46]. Similar to CONTRACTOR++, it also makes use of inferred value-based invariants to construct a better FSA from execution traces. SEKT first runs Daikon to infer value-based invariants and then runs a variant of k-tails [16] that utilizes the inferred invariants. Similar to k-tails, SEKT also requires that every two states that are merged together have the

same sequences of the next k invocations. However, different from k -tail, SEKT also requires that the merged states must share the same value-based invariants. This additional merging requirement allows SEKT to avoid problematic merges. In our study, we consider two variants of SEKT with its parameter k set to 1 and 2. The two variants are referred to as *SEKT 1-tails* and *SEKT 2-tails*.

- **TEMI:** Trace-enhanced MTS¹ inference (TEMI) is yet another recently proposed algorithm by Krka et al. [46]. TEMI has two main phases. In the first phase, TEMI runs an algorithm similar to CONTRACTOR++ to build a FSA. It considers transitions in the FSA built in the first phase as *maybe* transitions. In the second phase, TEMI converts *maybe* transitions that are observed in the execution traces to *required* transitions. TEMI has two variants: optimistic (it outputs all maybe and required transitions) and pessimistic (it outputs only required transitions).

Beschastnikh et al. have proposed an approach to specify FSA inference algorithms declaratively [14]. A specification consists of a set of property types (variable-labeled FSAs) and a composition function. Property instances matching the property type are mined from the traces (resulting in event-labeled FSAs) and these are then composed using the composition function into one FSA. The composition function resembles our model fusion step. However, we have a fundamentally different goal — to synergize existing model inference algorithms, rather than to describe existing or new inference algorithms. As a result, in our work we mine property instances that match the prescribed templates from the inferred models, rather than from the input traces. Additionally, our templates and fusion step are less generic than their property types and composition function as their aim is to express a variety of FSA inference algorithms.

¹Modal Transition System

Lo et al. propose an approach named SMArTIC that infers a finite state automaton from a set of execution traces [26]. This approach is built on a variant of k-tails automaton learning method that infers a probabilistic FSA and employs trace filtering and clustering. Erroneous traces are removed from the input execution traces and rather than learning a model from all the traces, the traces are clustered into groups and a separate FSA is learned from each group. These FSAs are later combined together into one FSA by identifying equivalent transitions – the goal is to get a larger FSA that accepts all the sentences accepted by the smaller FSAs. This is similar to the model fusion step of SpecForge. However, the aim of our work is to combine models from multiple specification mining algorithms.

Walkinshaw and Bogdanov propose an approach that allows users to manually input temporal properties to guide a specification mining algorithm in the inference of a FSA from execution traces [108]. This work was extended by Lo et al. who proposed an approach to automatically mine temporal properties from execution traces, and use these mined properties to automatically guide or steer a specification mining algorithm in its inference process [66]. As one step of SpecForge, we also infer temporal properties as constraints. However, rather than inferring them from execution traces, we infer them from FSAs that are generated by the underlying FSA mining algorithms on top of which SpecForge is built on. We do not use a data mining process to infer these properties, but rather a model checking algorithm.

Lorenzoli et al. [70], and Mariani and Pastore [76] propose two approaches named gkTail and KLFA to mine extended FSAs that incorporate data flow information. gkTail is able to infer algebraic constraints which specify restrictions on the values of some variables/arguments in the transitions of the FSAs. KLFA includes universally quantified constraints in the transitions of the FSAs to specify the re-occurrence of data values. Walkinshaw et al. have recently proposed an

approach to generate algebraic constraints for transitions in a FSA by leveraging a classification algorithm [109]. In this work, we focus on the generation of simple FSAs without algebraic constraints and quantified constraints.

SpecForge - Key Differences. None of the above mentioned approaches are able to combine multiple FSAs mined by different algorithms together. To our knowledge, SpecForge is the first approach that is capable of integrating the above mentioned miners: they can be used to learn new FSAs which can then be used as input to SpecForge. The goal of our work is to synergize many existing miners to build a more effective miner.

DSM - Key Differences. None of previously mentioned mining approaches employs deep learning for mining specifications. To the best of our knowledge, DSM (see Chapter 6) is the first work to use Recurrent Neural Network Language Models for mining automaton based software specifications. Our approach only processes execution traces to infer FSAs rather than considering program invariants as many previous approaches do (e.g., SEKT, CONTRACTOR++, TEMI [46], etc.). In comparison between SpecForge and DSM, SpecForge is a meta-approach that only accepts finite state automata and combine them together. On the other hand, DSM analyzes execution traces and use deep learning to mine a FSA from the input traces.

2.2.2 Rule-based Specification Mining.

There are several existing approaches that infer rules from program execution traces. Yang et al. propose Perracotta that mines two-event temporal rules from execution traces. To infer these rules, Perracotta uses a set of predefined rule templates and partitions input traces to several sub-traces. It computes satisfaction rate of a template, which is the number of partitions satisfying the template divided by the number of total partitions [119]. Lo et al. extend

Yang et al.’s approach by inferring from execution traces temporal rules with arbitrary lengths instead of two-event rules [64]. Li et al. also extend Yang et al.’s work by extracting simple linear temporal logic (LTL) rules from execution traces for hardware design [56]. Gruska et al. extract temporal properties of API usage and employ these properties to detect anomalies that deviate from the 6,000 projects [34]. Lo et al. mine length-2 quantified temporal rules which specify data-flow dependency constraints between method invocations [68]. Lo et al. also infer rules following the concept of Live Sequence Charts (LCSs), which are enriched with Daikon-style constraints [65]. Le and Lo investigate the effectiveness of several interestingness measures from data mining community for inferring rule-based specifications. Their findings indicate that other measures besides support and confidence can better detect correct two-event temporal rules from execution traces [48]. Lemieux et al. introduce Texada that mines temporal specifications in the form of linear temporal logic (LTL) of arbitrary length and complexity [55].

Key Differences. Different from above studies, all proposed specification mining approaches in this dissertation (i.e., SpecForge and DSM) outputs the final specifications in the form of FSAs. Among SpecForge and DSM, SpecForge is the only miner that infers temporal rules from input FSAs in its *model fission* process. SpecForge can potentially be integrated with the above mentioned studies, especially studies that mine temporal properties. These temporal properties can be combined with constraints that we infer from the mined FSAs and can be used to mine a more accurate FSA.

Chapter 3

INFORMATION RETRIEVAL AND SPECTRUM BASED BUG LOCALIZATION: BETTER TOGETHER

To deal with the limitation of existing techniques, in this work, I propose a new *hybrid-based* technique that *synthesizes* both bug reports and program spectra to localize bugs. My approach *adaptively* creates a bug-specific model to map a particular bug to its possible location, and introduces a novel idea of *suspicious words* that are highly associated to a bug.

3.1 Introduction

Existing bug localization approaches[95, 104, 127, 101, 40, 6, 96, 122, 124, 20, 58, 57, 60] only consider one kind of symptom or one source of information, i.e., only bug reports or only execution traces. This is a limiting factor since hints of the location of a bug may be spread in both bug report and execution traces; and some hints may only appear in one but not the other. In this work, I plan to address the limitation of existing studies by analyzing both bug reports and execution traces. I refer to the problem as *multi-modal bug localization* since I need to consider multiple modes of inputs (i.e., bug reports and program spectra). It fits well to developer debugging activities as illustrated by the following debugging scenarios:

1. Developer D is working on a bug report that is submitted to Bugzilla. One of the first tasks that he needs to do is to replicate the bug based on the description in the report. If the bug cannot be replicated, he will mark the bug report as “WORKSFORME” and will not continue further [82]. He will only proceed to the debugging step after the bug has been successfully replicated. After D replicates the bug, he has one or a few failing execution

traces. He also has a set of regression tests that he can run to get successful execution traces. Thus, after the replication process, D has *both* the textual description of the bug and a program spectra that characterizes the bug. With this, D can proceed to use multi-modal bug localization.

2. Developer D runs a regression test suite and some test cases fail. Based on his experience, D has some idea why the test cases fail. D can create a textual document describing the bug. At the end of this step, D has *both* program spectra and textual bug description, and can proceed to use multi-modal bug localization which will leverage not only the program spectra but also D 's domain knowledge to locate the bug.

Although no multi-modal bug localization technique has been proposed in the literature, there are a few multi-modal feature location techniques. These techniques process both feature description and program spectra to recommend program elements (typically program methods) that implement a corresponding feature [92, 61, 28]. These feature location approaches can be adapted to locate buggy program elements by replacing feature descriptions with bug reports and feature spectra with buggy program spectra. Unfortunately, my experiment (see Section 3.3) shows that the performance of such adapted approaches are not optimal yet.

My multi-modal bug localization approach improves previous multi-modal approaches based on two intuitions. First, I note that there are a wide variety of bugs [107, 115] and different bugs often require different treatments. Thus, there is a need for a bug localization technique that is adaptive to different types of bugs. Past approaches [61, 28, 92] propose a one-size-fits-all solution. Here, I propose an instance-specific solution that considers each bug individually and tunes various parameters based on the characteristic of the bug. Second, Parnin and Orso [88] highlight in their study that some words are useful in localizing bugs and suggest that “future research could also investigate ways to automatically suggest

or highlight terms that might be related to a failure”. Based on their observation, I design an approach that can automatically highlight *suspicious words* and use them to localize bugs.

My proposed approach, named Addaptive Multi-modal bug Localization (AML), realizes the above mentioned intuitions. It consists of three components: AML^{Text} , AML^{Spectra} , and AML^{SuspWord} . AML^{Text} only considers the textual description in bug reports, and AML^{Spectra} only considers program spectra. On the other hand, AML^{SuspWord} takes into account suspicious words learned by analyzing textual description and program spectra together. AML^{SuspWord} computes the *suspicious scores* of words that appear as comments or identifiers of various program elements. It associates a program element to a set of words and the suspiciousness of a word can then be computed based on the number of times the corresponding program elements appear in failing or correct execution traces. Each of these components would output a score for each program element, and AML computes the weighted sum of these scores. The final score is *adaptively* computed for each individual bug by tuning these weights.

I focus on localizing a bug to the method that contains it [85, 92, 118]. Historically, most IR-based bug localization techniques find buggy files [95, 104, 127, 101], while most spectrum-based bug localization solutions find buggy lines [40, 6, 96]. Localizing a bug to the file that contains it is useful, however a file can be big and developers still need to go through a lot of code to find the few lines that contain the bug. Localizing a bug to the line that contains it is useful, however, a bug often spans across multiple lines. Furthermore, developers often do not have “perfect bug understanding” [88] and thus by just looking at a line of code, developers often cannot determine whether it is the location of the bug and/or understand the bug well enough to fix it. A method is not as big as a file, but it often contains sufficient context needed to help developers understand a bug.

3.2 Adaptive Multi-modal Bug Localization

The overall framework of my Aaptive Multi-modal bug Localization (AML) is shown in Figure 6.1. AML (enclosed in dashed box) takes as input a new bug report and the program spectra corresponding to it. AML also takes as input a training set of (historical) bugs that have been localized before. For each bug in the training set, I have its bug report, program spectra, and set of faulty methods. If a method contains a root cause of the bug, it is labeled as *faulty*, otherwise it is labeled as *non-faulty*. Based on the training set of previously localized bugs and a method corpus, AML produces a list of methods ranked based on their likelihood to be the faulty ones given the new bug report.

AML has four components: AML^{Text} , AML^{Spectra} , AML^{SuspWord} , and Integrator. AML^{Text} processes only the textual information in the input bug reports using an IR-based bug localization technique described in Section 2.1.2. AML^{Text} in the end outputs a score for each method in the corpus. Given a bug report b and a method m in a corpus C , AML^{Text} outputs a score that indicates how close is m to b which is denoted as $AML^{\text{Text}}(b, m, C)$. By default, AML^{Text} uses VSM as the IR-based bug localization technique.

AML^{Spectra} processes only the program spectra information using a spectrum-based bug localization technique described in Section 2.1.1. AML^{Spectra} in the end outputs a score for each method in the corpus. Given a program spectra p and a method m in a corpus C , AML^{Spectra} outputs a score that indicates how suspicious is m considering p which is denoted as $AML^{\text{Spectra}}(p, m, C)$. By default, AML^{Spectra} uses Tarantula as the spectrum-based bug localization technique.

AML^{SuspWord} processes both bug reports and program spectra, and computes the suspiciousness scores of words to rank methods. Given a bug report b , a program spectra p , and a method m in a corpus C , AML^{SuspWord} outputs a score that indicates how suspicious is m considering b and p ; this is denoted as $AML^{\text{SuspWord}}(b, p, m, C)$.

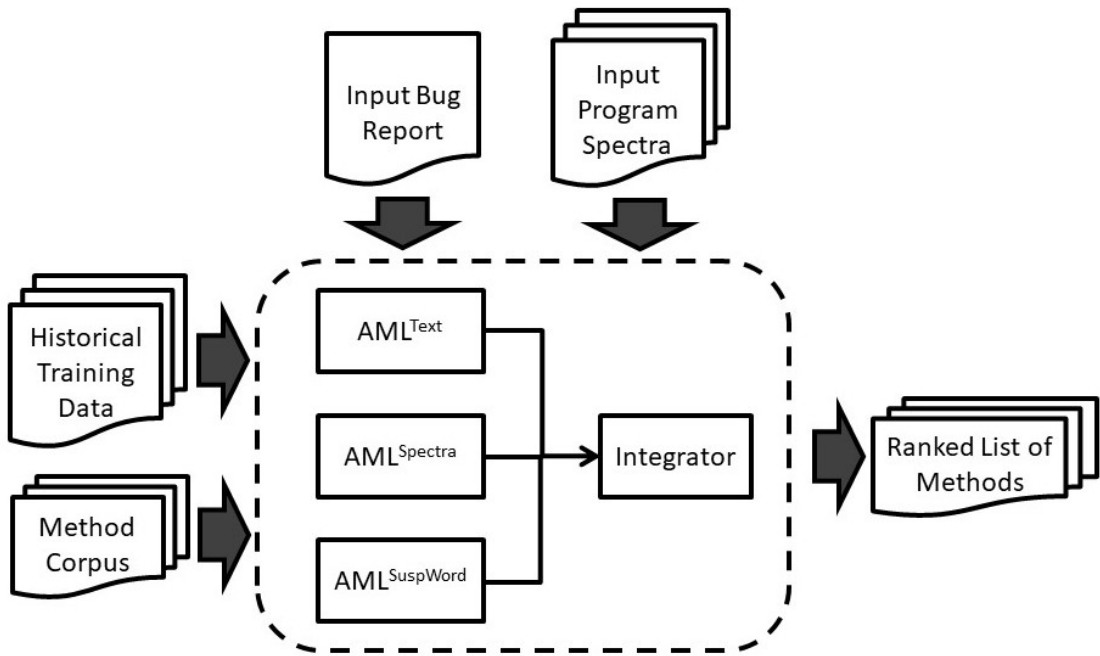


Figure 3.1: AML’s Framework

The integrator component combines the AML^{Text} , AML^{Spectra} , AML^{SuspWord} components to produce the final ranked list of methods. Given a bug report b , a program spectra p , and a method m in a corpus C , the adaptive integrator component outputs a suspiciousness score for method m which is denoted as $AML(b, p, m, C)$.

The AML^{Text} and AML^{Spectra} components reuse techniques proposed in prior works which are described in Section 2.1. In the next subsections, I just describe the new components namely AML^{SuspWord} and the adaptive integrator component.

3.2.1 Suspicious Word Component

Parnin and Orso highlighted that “future research could also investigate ways to automatically suggest or highlight terms that might be related to a failure” [88], however they did not propose a concrete solution. I use Parnin and Orso’s observation, which highlights that some words are indicative to the location of a bug, as a starting point to design my AML^{SuspWord} component. This component breaks

down a method into its constituent words, computes the suspiciousness scores of these words, and composes these scores back to result in the suspiciousness score of the method. The process is analogous to a machine learning or classification algorithm that breaks a data point into its constituent features, assign weights or importance to these features, and use these features, especially important ones, to assign likelihood scores to the data point. The component works in three steps: mapping of methods to words, computing word suspiciousness, and composing word suspiciousness into method suspiciousness. I describe each of these steps in the following paragraphs.

Step 1: Mapping of Methods to Words In this step, I map a method to its constituent words. For every method, I extract the following textual contents including: (1) The name of the method, along with the names of its parameters, and identifiers contained in the method body; (2) The name of the class containing the method, and the package containing the class; (3) The comments that are associated to the method (e.g., the Javadoc comment of that method, and the comments that appear inside the method), and comments that appear in the class (containing the method) that are not associated to any particular method.

After I have extracted the above textual contents, I apply the text pre-processing step described in Section 2.1.2. At the end of this step, for every method I map it to a set of pre-processed words. Given a method m , I denote the set of words it contains as $words(m)$.

Step 2: Computing Word Suspiciousness I compute the suspiciousness score of a word by considering the program elements that contain the word. Let us denote the set of all failing execution traces in spectra p as $p.F$ and the set of all successful execution traces as $p.S$. To compute the suspiciousness scores of a

word w given spectra p , I define several sets:

$$EF(w, p) = \{t \in p.F \mid \exists m \in t \text{ s.t. } w \in \text{words}(m)\}$$

$$ES(w, p) = \{t \in p.S \mid \exists m \in t \text{ s.t. } w \in \text{words}(m)\}$$

The set $EF(w, p)$ is the set of execution traces in $p.F$ that contain a method in which the word w appears. The set $ES(w, p)$ is the set of execution traces in $p.S$ that contain a method in which the word w appears. Based on these sets, I can compute the suspiciousness score of a word w using a formula similar to Tarantula as follows:

$$\text{SS}_{\text{word}}(w, p) = \frac{\frac{|EF(w, p)|}{|p.FAIL|}}{\frac{|EF(w, p)|}{|p.FAIL|} + \frac{|ES(w, p)|}{|p.SUCCESS|}} \quad (3.1)$$

Using the above formula, words that appear more often in methods that are executed in failing execution traces are deemed to be more suspicious than those that appear less often in such methods.

Step 3: Computing Method Suspiciousness To compute a method m 's suspiciousness score, I compute the textual similarity between m and the input bug report b , and consider the appearances of m in the input program spectra p . In the textual similarity computation, the suspiciousness of words are used to determine their weights.

First, I create a vector of weights that represents a bug report and another vector of weights that represents a method. Each element in a vector corresponds to a word that appears in either the bug report or the method. The weight of a word w in document (i.e., bug report or method) d of method corpus C considering program spectra p is:

$$\begin{aligned} \text{SSTFIDF}(w, p, d, C) = & \text{SS}_{\text{word}}(w, p) \times \log(f(w, d) + 1) \\ & \times \log \frac{|C|}{|d_i \in C : w \in d_i|} \end{aligned}$$

In the above formula, $SS_{\text{word}}(w, p)$ is the suspiciousness score of word w computed by Equation 3.1, $f(w, d)$ is the number of times word w appears in document d , and $d_i \in C$ means document d_i is in the set of document C . Similarly, $w \in d_i$ means word w belongs to document d_i . The above formula considers the weight of a word based on its suspiciousness, and well-known information retrieval metrics: term frequency (i.e., $\log(f(w, d) + 1)$) and inverse document frequency (i.e., $\log \frac{|C|}{|d_i \in C: w \in d_i|}$).

After the two vectors of weights of method m and bug report b are computed, I compute the suspiciousness score of the method m by computing the cosine similarity of these two vectors multiplied by a weighting factor. The formula to compute this score is as follows:

$$\begin{aligned}
 \text{AML}^{\text{SuspWord}}(b, p, m, C) &= SS_{\text{method}}(m, p) \times \\
 &\frac{\sum_{w \in b \cap m} \text{SSTFIDF}(w, p, b, C) \times \text{SSTFIDF}(w, p, m, C)}{\sqrt{\sum_{w \in b} \text{SSTFIDF}(w, p, b, C)^2} \times \sqrt{\sum_{w \in m} \text{SSTFIDF}(w, p, m, C)^2}} \quad (3.2)
 \end{aligned}$$

Here I use $SS_{\text{method}}(m, p)$ that computes the suspiciousness score of method m considering program spectra p as the weighting factor. This can be computed by various spectrum-based bug localization tools. By default, I use the same fault localization tool as the one used in $\text{AML}^{\text{Spectra}}$ component. With this, $\text{AML}^{\text{SuspWord}}$ integrates both macro view of method suspiciousness (which considers direct execution of a method in the failing and correct execution traces) and micro view of method suspiciousness (which considers the executions of its constituent words in the execution traces).

3.2.2 Integrator Component

The integrator component serves to combine the scores produced by the three components AML^{Text} , $\text{AML}^{\text{Spectra}}$ and $\text{AML}^{\text{SuspWord}}$ by taking a weighted sum of the scores. The final suspiciousness score of method m given bug report b and

program spectra p in a corpus C is given by:

$$\begin{aligned}
 f(x_i, \theta) = & \alpha \times \text{AML}^{\text{Text}}(b, m) + \beta \times \text{AML}^{\text{Spectra}}(p, m) \\
 & + \gamma \times \text{AML}^{\text{SuspWord}}(b, p, m)
 \end{aligned}
 \tag{3.3}$$

where i refers to a specific (b, p, m) combination (aka *data instance*), x_i denotes the feature vector $x_i = [\text{AML}^{\text{Text}}(b, m), \text{AML}^{\text{Spectra}}(p, m), \text{AML}^{\text{SuspWord}}(b, p, m)]$, and θ is the parameter vector $[\alpha, \beta, \gamma]$, where α, β, γ are arbitrary real numbers. Note that I exclude mentioning corpus C in both sides of Equation 3.3 to simplify the set of notations used in this section.

The weight parameters (θ) are *tuned adaptively* for a new bug report b based on a set of top-K historical fixed bugs in a training data that are the most similar to b . I find these top-K nearest neighbors by measuring the textual similarity of b with training (historical) bug reports using the VSM model. In this work, I propose a probabilistic learning approach which analyzes this training data to fine-tune the weight parameters α , β , and γ for the new bug report b . The selection of top-K helps filter noise and less useful features from fixed bugs which are irrelevant to bug report b . That assists the tuning process of α , β , and γ to find the best weights to combine scores of three components of AML. Note that if the value of K is set to too small, α , β , and γ are potentially assigned to inaccurate weights as there are too little information from fixed bugs are used for learning weights.

Probabilistic Formulation From a machine learning standpoint, bug localization can be interpreted as a *binary classification* task. For a given combination (b, p, m) , the positive label refers to the case when method m is indeed where the bug b is located (i.e., faulty case), and the negative label is when m is not relevant to b (i.e., non-faulty case). As I deal with binary classification task, it is plausible

to assume that a data instance follows Bernoulli distribution, c.f., [83]:

$$p(x_i, y_i | \theta) = \sigma(f(x_i, \theta))^{y_i} (1 - \sigma(f(x_i, \theta)))^{(1-y_i)} \quad (3.4)$$

where $y_i = 1$ ($y_i = 0$) denotes the positive (negative) label, and $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the logistic function. Using this notation, I can formulate the overall data *likelihood* as:

$$p(X, y | \theta) = \prod_{i=1}^N \sigma(f(x_i, \theta))^{y_i} (1 - \sigma(f(x_i, \theta)))^{(1-y_i)} \quad (3.5)$$

where N is the total number of data instances (i.e., (b, p, m) combinations), and $y = [y_1, \dots, y_i, \dots, y_N]$ is the label vector.

My primary interest here is to infer the *posterior* probability $p(\theta|X)$, which can be computed via the Bayes' rule:

$$p(\theta|X, y) = \frac{p(X, y|\theta)p(\theta)}{p(X, y)} \quad (3.6)$$

Specifically, my goal is to find an optimal parameter vector θ^* that maximizes the posterior $p(\theta|X, y)$. This leads to the following optimization task:

$$\begin{aligned} \theta^* &= \arg \max_{\theta} p(\theta|X, y) \\ &= \arg \max_{\theta} p(X, y|\theta)p(\theta) \\ &= \arg \min_{\theta} (-\log(p(X, y|\theta)) - \log(p(\theta))) \end{aligned} \quad (3.7)$$

Here I can drop the denominator $p(X, y)$, since it is independent of the parameters θ . The term $p(\theta)$ refers to the *prior*, which I define to be a Gaussian distribution with (identical) zero mean and inverse variance λ :

$$p(\theta) = \prod_{j=1}^J \sqrt{\frac{\lambda}{2\pi}} \exp\left(-\frac{\lambda}{2}\theta_j^2\right) \quad (3.8)$$

where the number of parameters J is 3 in my case (i.e., α , β , and γ).

By substituting (3.5) and (3.8) into (3.7), and by dropping the constant terms that are independent of θ , the optimal parameters θ^* can be computed as:

$$\theta^* = \arg \min_{\theta} \left(\sum_{i=1}^N \mathcal{L}_i + \frac{\lambda}{2} \sum_{j=1}^J \theta_j^2 \right) \quad (3.9)$$

where \mathcal{L}_i is called the instance-wise loss, as given by:

$$\mathcal{L}_i = - [y_i \log(\sigma(f(x_i, \theta))) + (1 - y_i) \log(1 - \sigma(f(x_i, \theta)))] \quad (3.10)$$

Solution to this minimization task is known as the *regularized logistic regression*. The regularization term $\frac{\lambda}{2} \sum_{j=1}^J \theta_j^2$ —which stems from the prior $p(\theta)$ —serves to penalize large parameter values, thereby reducing the risk of data overfitting.

Algorithm To estimate θ^* , I develop an iterative parameter tuning strategy that performs a descent move along the negative gradient of \mathcal{L}_i . Algorithm 1 summarizes my proposed parameter tuning method. More specifically, for each instance i , I perform gradient descent update for each parameter θ_j :

$$\theta_j \leftarrow \theta_j - \eta \left(\frac{\partial \mathcal{L}_i}{\partial \theta_j} + \lambda \theta_j \right) \quad (3.11)$$

where the gradient term $\frac{\partial \mathcal{L}_i}{\partial \theta_j}$ resolves to:

$$\frac{\partial \mathcal{L}_i}{\partial \theta_j} = (\sigma(f(x_i, \theta)) - y_i) x_{i,j} \quad (3.12)$$

with the feature values $x_{i,1} = \text{AML}^{\text{Text}}(b, m)$, $x_{i,2} = \text{AML}^{\text{Spectra}}(p, m)$, and $x_{i,3} = \text{AML}^{\text{SuspWord}}(b, p, m)$, corresponding to the parameters α , β and γ , respectively. The update steps are realized in lines 11-13 of Algorithm 1.

One key challenge in the current bug localization task is the extremely skewed distribution of the labels, i.e., the number of positive cases is much smaller than

Algorithm 1 Iterative parameter tuning

Require: Matrix $X \in \mathbb{R}^{N \times 3}$ (each row is a vector $x_i = [\text{AML}^{\text{Text}}(b, m), \text{AML}^{\text{Spectra}}(p, m), \text{AML}^{\text{SuspWord}}(b, p, m)]$ for bug report b , program spectra p , and method m in one of the top-K most similar training data), label vector $y \in \mathbb{R}^N$ (each element y_i is the label of x_i), learning rate η , regularization parameter λ , maximum training iterations T_{max}

Ensure: Weight parameters α, β, γ

```
1: Initialize  $\alpha, \beta, \gamma$  to zero
2: repeat
3:   for each  $n \in \{1, \dots, N\}$  do
4:     if  $n \bmod 2 = 0$  then ▷ Draw a positive instance
5:       Randomly pick  $i$  from  $\{1, \dots, N\}$  s.t.  $y_i = 1$ 
6:     else ▷ Draw a negative instance
7:       Randomly pick  $i$  from  $\{1, \dots, N\}$  s.t.  $y_i = 0$ 
8:     end if
9:     Compute overall score  $f(x_i, \theta)$  using Eq. (3.3)
10:    Compute gradient  $g_i \leftarrow \sigma(f(x_i, \theta)) - y_i$ 
11:     $\alpha \leftarrow \alpha - \eta (g_i \times \text{AML}^{\text{Text}}(b, m) + \lambda\alpha)$ 
12:     $\beta \leftarrow \beta - \eta (g_i \times \text{AML}^{\text{Spectra}}(p, m) + \lambda\beta)$ 
13:     $\gamma \leftarrow \gamma - \eta (g_i \times \text{AML}^{\text{SuspWord}}(b, p, m) + \lambda\gamma)$ 
14:  end for
15: until  $T_{max}$  iterations
```

the number of negative cases. To address this, I devise a *balanced random sampling* procedure when picking a data instance for gradient descent update. In particular, for every update step, I alternately select a random instance from the positive and negative instance pools, as per lines 4-8 of Algorithm 1.

Using this simple method, I can balance the training from positive and negative instances, thus effectively mitigating the issue of *skewed distribution* in the localization task. It is also worth noting that my iterative tuning procedure is efficient. That is, its time complexity is linear with respect to the number of instances N and maximum iterations T_{max} .

Table 3.1: Dataset Description

Project	#Bugs	Time Period	Average Number of Methods
AspectJ	41	03/2005 – 02/2007	14,218.39
Ant	53	12/2001 – 09/2013	9,624.66
Lucene	37	06/2006 – 01/2011	10,220.14
Rhino	26	12/2007 – 12/2011	4,839.58

3.3 Evaluation

3.3.1 Methodology

Dataset I use a dataset of 157 bugs from 4 popular software projects to evaluate my approach against the baselines. These projects are AspectJ [3], Ant [1], Lucene [2], and Rhino [5]. All four projects are medium-large scale and implemented in Java. AspectJ, Ant, and Lucene contain more than 300 kLOC and at least 9000 methods, while Rhino contains almost 100 kLOC and approximately 5000 methods. Table 3.1 describes detailed information of the four projects in my study.

The 41 AspectJ bugs are from the iBugs dataset which were collected by Dallmeier and Zimmermann [24]. Each bug in the iBugs dataset comes with the code before the fix (pre-fix version), the code after the fix (post-fix version), and a set of test cases. The iBugs dataset contains more than 41 AspectJ bugs but not all of them come with failing test cases. Test cases provided in the iBugs dataset are obtained from the various versions of the regression test suite that comes with AspectJ. The remaining 116 bugs from Ant, Lucene, and Rhino are collected by ourselves following the procedure used by Dallmeier and Zimmermann [24]. For each bug, I collected the pre-fix version, post-fix version, a set of successful test cases, and at least one failing test case. A failing test case is often included as an attachment to a bug report or committed along with the fix in the post-fix version. When a developer receives a bug report, he/she first needs to replicate the error described in the report [82]. In this process, he is creating a failing test

case. Unfortunately, not all test cases are documented and saved in the version control systems.

Evaluation Metric and Settings I use two metrics namely mean average precision (MAP) and Top N to evaluate the effectiveness of a bug localization solution. They are defined as follows:

- **Top N:** Given a bug, if one of its faulty methods is in the top-N results, I consider the bug is successfully localized. Top N score of a bug localization tool is the number of bugs that the tool can successfully localize [127, 101].
- **Mean Average Precision (MAP):** MAP is an IR metric to evaluate ranking approaches [75]. MAP is computed by taking the mean of the *average precision* scores across all bugs. The average precision of a single bug is computed as:

$$AP = \sum_{k=1}^M \frac{P(k) \times pos(k)}{\text{number of buggy methods}}$$

where k is a rank in the returned ranked methods, M is the number of ranked methods, and $pos(k)$ indicates whether the k^{th} method is faulty or not. $P(k)$ is the precision at a given top k methods and is computed as follows:

$$P(k) = \frac{\# \text{faulty methods in the top } k}{k}.$$

Note that typical MAP scores of existing bug localization techniques are low [95, 104, 127, 101].

I use 10 fold cross validation: for each project, I divide the bugs into ten sets, and use 9 as training data and 1 as testing data. I repeat the process 10 times using different training and testing data combinations. I then aggregate the results to get the final Top N and MAP scores. The learning rate η and regularization

parameter λ of AML are chosen by performing another cross validation on the training data, while the maximum number of iterations T_{max} is fixed as 30. I use $K = 10$ as default value for the number of nearest neighbors. I conduct experiments on an Intel(R) Xeon E5-2667 2.9GHz server running Linux 2.6.

I compare my approach against 3 state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [92], DIT^A and DIT^B [28]), a state-of-the-art IR-based bug localization technique named LR [120], and a state-of-the-art spectrum-based bug localization technique named MULTRIC [118]. I use the same parameters and settings that are described in their papers with the following exceptions that I justify. For DIT^A and DIT^B, the threshold used to filter methods using HITS was decided “such that at least one gold set method remained in the results for 66% of the [bugs]” [28]. In this paper, since I use ten-fold cross validation, rather than using 66% of all bugs, I use all bugs in the training data (i.e., 90% of all bugs) to tune the threshold. For PROMESIR, I also use 10-fold CV and apply a brute force approach to tune PROMESIR’s component weights using a step of 0.05. PROMESIR, DIT^A, DIT^B, and MULTRIC locate buggy methods, however LR locate buggy files. Thus, I convert the list of files that LR produces into a list of methods by using two heuristics: (1) return methods in a file in the same order that they appear in the file; (2) return methods based on their similarity to the input bug report as computed using a VSM model. I refer to the two variants of LR as LR^A and LR^B respectively.

3.3.2 RQ1: AML vs. Baselines

PROMESIR [92], SITIR [61], and several algorithm variants proposed by Dit et al. [28] are state-of-the-art multi-modal feature location techniques. Among the variants proposed by Dit et al. [28], the best performing ones are $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. I refer to them as DIT^A and DIT^B in this paper. Dit et al. have shown that

Table 3.2: Top N: AML vs. Baselines. $N \in \{1, 5, 10\}$. Numbers in parentheses indicate percentages of successfully localized bugs among all of 157 bugs.

Top	Project	AML	PROMESIR	DIT ^A	DIT ^B	LR ^A	LR ^B	MULTRIC
1	AspectJ	7 (4.46%)	4 (2.55%)	4 (2.55%)	3 (1.91%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Ant	9 (5.73%)	7 (4.46%)	3 (1.91%)	3 (1.91%)	1(0.64%)	11 (7.01%)	2 (1.27%)
	Lucene	11 (7.01%)	8 (5.10%)	7 (4.46%)	7 (4.46%)	1 (0.64%)	7 (4.46%)	4 (2.55%)
	Rhino	4 (2.55%)	2 (1.27%)	1 (0.64%)	1 (0.64%)	0 (0.00%)	2 (1.27%)	2 (1.27%)
	Overall	31 (19.75%)	21 (13.28%)	15 (9.55%)	14 (8.92%)	2 (1.27%)	20 (12.74%)	8 (5.10%)
5	AspectJ	13 (8.28%)	6 (3.82%)	4 (2.55%)	3 (1.91%)	0 (0.00%)	0 (0.00%)	1 (0.64%)
	Ant	22 (14.01%)	17 (10.83%)	10 (6.37%)	10 (6.37%)	11 (7.01%)	20 (12.74%)	7 (4.46%)
	Lucene	22 (14.01%)	18 (11.46%)	13 (8.28%)	13 (8.28%)	6 (3.82%)	16 (10.19%)	13 (8.28%)
	Rhino	14 (8.92%)	13 (8.28%)	5 (3.18%)	5 (3.18%)	2 (1.27%)	8 (5.10%)	8 (5.10%)
	Overall	71 (45.22%)	54 (34.40%)	32 (20.38%)	31 (19.75%)	19 (12.10%)	44 (28.02%)	29 (18.47%)
10	AspectJ	13 (8.28%)	9 (5.73%)	4 (2.55%)	3 (1.91%)	0 (0.00%)	0 (0.00%)	2 (1.27%)
	Ant	31 (19.75%)	28 (17.83%)	20 (12.74%)	20 (12.74%)	19 (12.10%)	32 (20.38%)	15 (9.55%)
	Lucene	29 (18.47%)	21 (13.38%)	20 (12.74%)	19 (12.10%)	10 (6.37%)	24 (15.29%)	16 (10.19%)
	Rhino	19 (12.10%)	14 (8.92%)	7 (4.46%)	7 (4.46%)	3 (1.91%)	12 (7.64%)	11 (7.01%)
	Overall	92 (58.60%)	72 (45.86%)	51 (32.48%)	49 (31.21%)	32 (20.38%)	68 (43.31%)	44 (28.02%)

Table 3.3: Mean Average Precision: AML vs. Baselines.

Project	AML	PROMESIR	DIT ^A	DIT ^B	LR ^A	LR ^B	MULTRIC
AspectJ	0.187	0.121	0.092	0.071	0.006	0.004	0.016
Ant	0.234	0.206	0.120	0.120	0.070	0.218	0.077
Lucene	0.284	0.204	0.169	0.166	0.063	0.184	0.188
Rhino	0.243	0.203	0.092	0.090	0.034	0.103	0.172
Overall	0.237	0.184	0.118	0.112	0.043	0.127	0.113



Figure 3.2: AML vs. Baselines in terms of Top N ($N \in \{1, 5, 10\}$) and MAP.

these two variants outperform SITIR. However, Dit et al.’s variants have never been compared with PROMESIR. PROMESIR has also never been compared with SITIR. Thus, to answer this research question, I compare the performance of my approach with PROMESIR, DIT^A and DIT^B. I also compare with the two variants of LR [120] (LR^A and LR^B) and MULTRIC [118] which are recently proposed state-of-the-art IR-based and spectrum-based bug localization techniques respectively. Table 3.2 and Figure 3.2 show the performance of AML, and all the baselines in terms of Top N. Out of the 157 bugs, AML can successfully localize 31, 71, and 92 bugs when developers inspect the top 1, top 5, and top 10 methods respectively. This means that AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs than the best baseline (i.e., PROMESIR) by investigating the top 1, top 5, and top 10 methods respectively.

Table 3.3 and Figure 3.2 show the performance of AML and the baselines in terms of MAP. AML achieves MAP scores of 0.187, 0.234, 0.284, and 0.243 for AspectJ, Ant, Lucene, and Rhino datasets, respectively. Averaging across the four projects, AML achieves an overall MAP score of 0.237 which outperforms all the baselines. AML improves the average MAP scores of PROMESIR, DIT^A, DIT^B, LR^A, LR^B, and MULTRIC by 28.80%, 100.85%, 111.61%, 451.16%, 91.34%, and 109.73% respectively. Moreover, considering each individual project, in terms of MAP, AML is still the best performing multi-modal bug localization approach. AML outperforms the MAP score of the best performing baseline, by 54.55%, 13.59%, 39.22%, and 19.70% for AspectJ, Ant, Lucene, and Rhino datasets, respectively.

Moreover, I find that my novel component of AML, i.e., AML^{SuspWord}, can outperform all the baselines. AML^{SuspWord} can achieve a Top 1, Top 5, Top 10, and MAP scores of 26, 66, 83, and 0.193. These results outperform the best performing baseline by 23.81%, 22.22%, 15.28%, and 4.89% respectively.

Table 3.4: Contributions of AML Components

Approach	Top 1	Top 5	Top 10	MAP
AML ^{-Text}	28	68	87	0.212
AML ^{-SuspWord}	28	62	83	0.201
AML ^{-Spectra}	26	63	84	0.210
AML	31	71	92	0.237

3.3.3 RQ2: Contributions of AML Components

To answer this research question, I simply drop one component (i.e., AML^{Text}, AML^{SuspWord}, and AML^{Spectra}) from AML one-at-a-time and evaluate their performance. In the process, I create three variants of AML: AML^{-Text}, AML^{-SuspWord}, and AML^{-Spectra}. To create AML^{-Text}, AML^{-SuspWord}, and AML^{-Spectra}, I exclude AML^{Text}, AML^{SuspWord}, and AML^{Spectra} components from Equation 3.3 of my proposed AML, respectively. I use the default value of $K = 10$, and apply Algorithm 1 to tune weights of these variants, and compare their performance with my proposed AML.

Table 3.4 and Figure 3.3 show the performance of the three AML variants, and the full AML. From the table, the full AML has the best performance in term of Top 1, Top 5, Top 10, and MAP. This shows that omitting one of the AML components reduces the effectiveness of AML. Thus, *each of the component contributes towards the overall performance of AML*. Also, among the variants, AML^{-SuspWord} has the smallest Top 1, Top 5, Top 10, and MAP scores. The reduction in the evaluation metric scores are the largest when I omit AML^{SuspWord}. This indicates that AML^{SuspWord} *is more important than the other components of AML*.

3.3.4 RQ3: Integrator vs. SVM^{rank}

Rather than using the integrator component, it is possible to use a standard machine learning algorithm, e.g., learning-to-rank, to combine the scores produced by

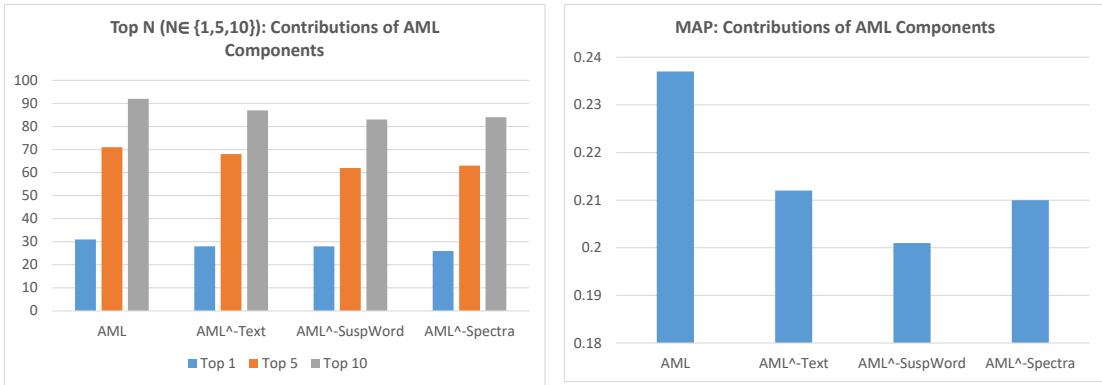


Figure 3.3: Contributions of AML Components in Bar Charts.

Table 3.5: Integrator vs. SVM^{rank} .

Metrics	Project	Integrator	SVM^{rank}
Top 1	AspectJ	7	4
	Ant	9	7
	Lucene	11	10
	Rhino	4	4
	Overall	31	25
Top 5	AspectJ	13	11
	Ant	22	24
	Lucene	22	23
	Rhino	14	13
	Overall	71	71
Top 10	AspectJ	13	14
	Ant	31	31
	Lucene	29	26
	Rhino	19	16
	Overall	92	87
MAP	AspectJ	0.187	0.131
	Ant	0.234	0.234
	Lucene	0.284	0.267
	Rhino	0.243	0.227
	Overall	0.237	0.215

AML^{Text}, AML^{SuspWord}, and AML^{Spectra}. Indeed, the two state-of-the-art IR-based and spectrum-based bug localization techniques (i.e., LR and MULTRIC) are based on learning-to-rank. In this research question, I want to compare my Integrator component with an off-the-shelf learning-to-rank tool namely SVM^{rank} [39], which was also used by LR [120]. I simply replace the Integrator component with

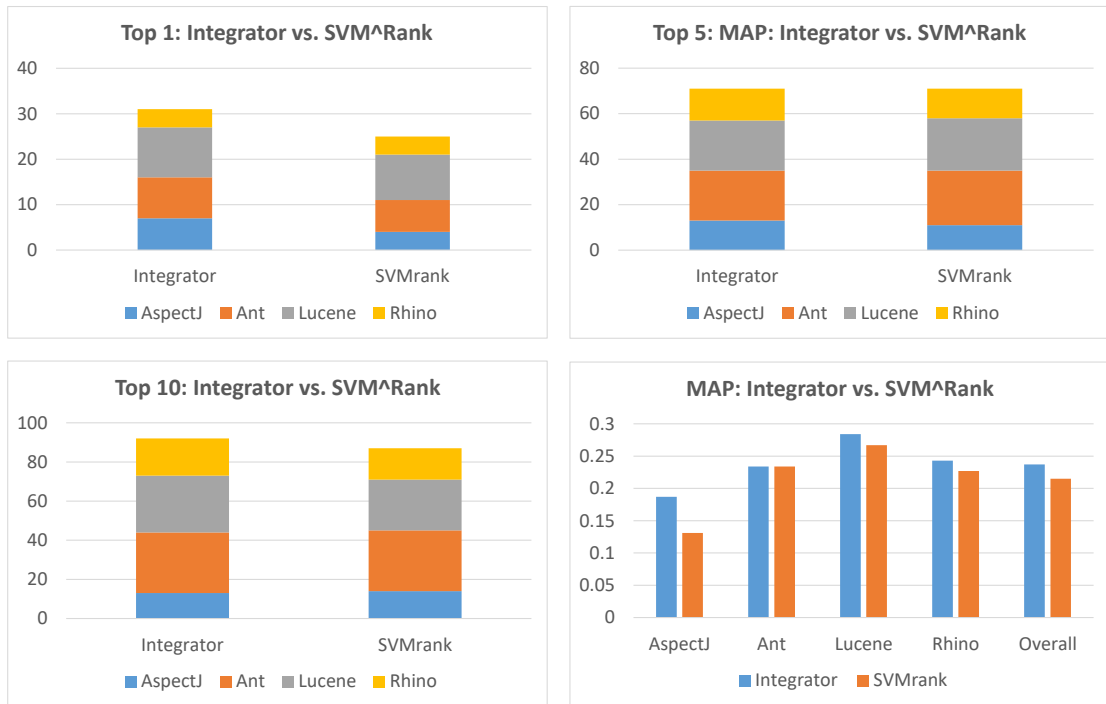


Figure 3.4: Integrator vs. SVM^{rank} in Bar Charts.

SVM^{rank} and evaluate the effectiveness of the resulting solution.

Table 3.5 and Figure 3.4 show the results of comparing my Integrator with SVM^{rank} . I can note that for most subject programs and metrics, Integrator outperforms SVM^{rank} . This shows the benefit of my Integrator component which builds a personalized model for each bug and considers the data imbalance phenomenon.

3.3.5 RQ4: Running Time

If AML takes hours to produce a ranked list of methods for a given bug report, then it would be less useful. In this research question, I investigate the average running time needed for AML to output a ranked list of methods for a given bug report. Table 4.8 shows means and standard deviations of AML’s running time for different projects. From the table, I note that AML has an average running time of 46.01 seconds. Among the four projects, AML can process Rhino bugs

Table 3.6: Running Time of AML (seconds)

Project	Mean	Standard Deviation
AspectJ	72.79	5.50
Ant	40.88	2.52
Lucene	43.39	3.40
Rhino	17.94	1.58
Overall	46.01	18.48

with the least average running time (i.e., 17.94 seconds), and AML needs the longest running time to process AspectJ bugs (i.e., 72.79 seconds). Compared to the other three projects, AspectJ is considerably larger. Therefore, it takes more time for AML to tune its component weights. Considering that a developer can spend hours and even days to fix a bug [42], AML running time of 20-80 seconds is reasonable.

3.3.6 RQ5: : Effect of Varying Number of Neighbors

Table 3.7: Effect of Varying Number of Neighbors (K)

#Neighbors	Top 1	Top 5	Top 10	MAP
$K = 5$	29	68	84	0.223
$K = 10$	31	71	92	0.237
$K = 15$	30	70	91	0.237
$K = 20$	29	70	88	0.227
$K = 25$	29	67	87	0.224
$K = \infty$	28	69	86	0.222

My proposed approach takes as input one parameter, which is the number of neighbors K , that is used to adaptively tune the weights α , β , and γ for a bug. By default, I set the number of neighbors to 10. The effect of varying this default value is unclear. To answer this research question, I vary the value of K and I investigate the effect of different numbers of neighbors on the performance of AML. In particular, I want to investigate if the performance of AML remains relatively stable for a wide range of K . To answer this research question, I vary the number of

neighbors K from 5 to all bugs in the training data (i.e., $K = \infty$). The results with varying numbers of neighbors is shown in Table 3.7. I can see that, as I increase K , the performance of AML increases until a certain point. When I use a large K , the performance of AML decreases. This suggests that in general including more neighbors can improve performance. However, an overly large number of neighbors may lead to an increased level of noise (i.e., the number of non-representative neighbors), resulting in a degraded performance. The differences in the Top N and MAP scores are small though.

3.3.7 Discussion

Number of Failed Test Cases and Its Impact. In my experiments with 157 bugs, most of the bugs come with few failed test cases (average = 2.185). I investigate whether the number of failed test cases impacts the effectiveness of my approach. I compute the differences between the average number of failed test cases for bugs that are successfully localized at top-N positions ($N = 1, 5, 10$) and bugs that are not successfully localized. I find that the differences are small (-0.472 to 0.074 test cases). These indicate that the number of test cases do not impact the effectiveness of my approach much and typically 1 to 3 failed test cases are sufficient for my approach to be effective.

AML’s Ineffective Cases. I manually analyze a number of cases where AML is unable to localize faulty methods with top-N positions ($N \in \{1, 5, 10\}$) of output ranked lists. I find that for many bugs, the tuned values of α , β , and γ are not the best ones. Therefore, faulty methods are not assigned to highest suspiciousness scores, and AML cannot localize root causes of these bugs in top-N positions ($N \in \{1, 5, 10\}$). There are many reasons to explain why learned values of α , β , and γ are not optimal. Firstly, that could be because AML only considers three types of features computed by the three components (i.e., AML^{Text} , $\text{AML}^{\text{Spectra}}$, and $\text{AML}^{\text{SuspWord}}$). The limited information from the current three features could

be a factor affecting AML. I believe adding more bug localization approaches (e.g., spectrum-based, IR-based bug localization, etc.) might improve the effectiveness of AML. Secondly, that could be our algorithm of tuning α , β , and γ is not robust enough to handle noises from training data. For the future work, I plan to utilize other approaches such as Newton’s method [12] to tune weights of AML components.

3.4 Conclusion

In this work, I put forward a novel multi-modal bug localization approach named Addaptive Multi-modal bug Localization (AML). Different from previous multi-modal approaches that are one-size-fits-all, my proposed approach can adapt itself to better localize each new bug report by tuning various weights learned from a set of training bug reports that are relevant to the new report. AML (in particular its $AML^{SuspWord}$ component) also leverages the concept of *suspicious words* (i.e., words that are associated to a bug) to better localize bugs. I have evaluated my proposed approach on 157 real bugs from 4 software systems. My experiments highlight that, among the 157 bugs, AML can successfully localize 31, 71, and 92 bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Compared to the best performing baseline, AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Furthermore, in terms of MAP, AML outperforms the best baseline by 28.80%.

In the future, I plan to improve the effectiveness of my proposed approach in terms of Top N and MAP scores. To reduce the threats to external validity, I also plan to investigate more bug reports from additional software systems.

Dataset. Additional information of the 157 bugs used in the experiments is available at <https://github.com/lebuitienduy/aml>.

Chapter 4

LEARNING-TO-RANK BASED FAULT LOCALIZATION USING LIKELY INVARIANTS

In this work, I propose *Savant*, a new spectrum-based fault localization approach that employs a learning-to-rank strategy, using likely invariant *diffs* and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. *Savant* has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. At the end of these four steps, *Savant* produces a short ranked list of potentially buggy methods.

4.1 Introduction

This work particularly focuses on a family of automated debugging solutions that takes as input a set of failing and passing test cases and then highlights the suspicious program elements that are likely responsible for the failures (failing test cases), e.g., [6, 10, 11, 20, 40, 57, 60, 71, 124, 122]. While these techniques have been shown effective in many contexts, their effectiveness needs to be further improved to localize more bugs more accurately.

I propose a novel technique, *Savant*, for effective automated fault localization. *Savant* uses a learning-to-rank machine learning approach to identify buggy methods from failures by analyzing both classic suspiciousness scores and inferred likely invariants observed on passing and failing test cases. *Savant* is built on three high-level intuitions. First, program elements which follow invariants of failing runs that are incompatible with invariants of correct runs are suspicious. Second, such program elements are even more suspicious if they are assigned higher suspiciousness scores computed by existing spectrum-based fault localization (SBFL)

formulas [7, 40, 117, 118, 121]. Third, some invariant differences are likely to be more suspicious than others. For example, the violation of a “non-zero” invariant (e.g., $x \neq 0$ or $x \neq \text{null}$) in the failing execution may indicate a division by zero or null pointer dereference, and is thus likely to be more suspicious than a violation of a “linear binary” invariant (e.g., $x + y + 3 = 0$), due to the prevalence of null pointer dereference errors. There exists natural variations in existing invariant learning techniques when applied to different executions. The challenge is thus to distinguish between invariant differences that arise from natural execution variation and those that are truly indicative of buggy behavior. In the absence of a clear *a priori* set of rules, I propose to learn the relative importance of different invariant differences and suspiciousness scores from pre-existing fixed bugs, and use that learned information to localize new bugs.

Savant is designed for efficiency and reliability, and employs a number of steps to both reduce runtime and make informed recommendations based on the inferred invariants and computed suspiciousness scores. *Savant* works at the method-level granularity [85, 92, 118] rather than file- [101, 127] or statement-level [117, 121]. Although localizing a bug to the file-level is useful, files can be large, leaving considerable code for developers to filter down to the few lines that contain a bug. For example, the faulty file corresponding to Bug 383 in the Closure Compiler (see Figure 4.1) is 1,160 lines of code in total. Localizing a bug to the line-level, on the other hand, is ill-suited for multi-line bugs, which are common [72]. Furthermore, developers often lack “perfect bug understanding” [88] and thus looking at a single line of code does not always allow a developer to determine whether it is truly buggy, nor to understand the bug well enough to fix it. A method is not as big as a file, but often contains sufficient context needed to help developers understand a bug. Furthermore, Kochhar et al. reported that 51.81% of their 386 survey respondents chose method-level as the preferred granularity for fault localization [44].

Savant consists of four steps: method clustering & test case selection, invariant learning, feature extraction, and method ranking. Unlike prior work, e.g., [6, 40, 71], *Savant* does not use all available test cases to localize faults. Instead, for scalability, it selects a subset of test cases, including both the failing tests and only those passing tests that cover similar program elements. Next, it uses Daikon [30] to learn likely invariants on the entry and exit of only those methods that are executed by the failing executions. By limiting the number of instrumented methods, Daikon completes its learning process much more quickly than it would by default, especially for larger systems. For efficiency reasons, I choose to *kill* Daikon if it does not complete its processing within one minute. I run Daikon in several settings: first, on traces collected from all executions; then only on correct executions; and finally only on failing executions. By *diff*-ing all pairs of resultant invariant sets, I identify suspicious methods where invariants inferred from one set do not hold in another. Next, I convert the invariant diffs into a set of features. I also use the suspiciousness scores computed by several SBFL formulae for the suspicious methods as features. All of the extracted features are then provided as input to a learning-to-rank algorithm. The learning-to-rank algorithm learns a ranking model based on a training set of fixed bugs which differentiates invariant differences of faulty and non-faulty methods. This ranking model can then be used to rank suspicious methods for new bugs based on their corresponding invariant differences.

4.2 Motivating Example

I begin by introducing an example defect to motivate my technique. Consider Bug 383 in the Closure Compiler’s bug database¹, summarized in the top part of Figure 4.1. The bug is assigned a high priority: it causes an issue in Internet Explorer 9 and `jQuery.getScript`. According to the developer patch (bottom of Figure 4.1), the bug ultimately resides in the `strEscape` method in the

¹<https://goo.gl/YtW6Ux>

<p>Bug 383 (Priority: high)</p> <p>Summary: \0 \x00 and \u0000 are translated to null character</p> <p>Description:</p> <p>What steps will reproduce the problem?</p> <p>1. write script with string constant “\0” or “\x00” or “\u0000”</p> <p>What is the expected output? What do you see instead?</p> <p>I expected a string literal with “\0” (or something like that) and instead get a string literal with three null character values.</p> <p>Please provide any additional information below.</p> <p>This is causing an issue with IE9 and jQuery.getScript. It causes IE9 to interpret the null character as the end of the file instead of a null character.</p>
<pre>@@ -963,6 +963,7 @@ class CodeGenerator { for (int i = 0; i < s.length(); i++) { char c = s.charAt(i); switch (c) { + case '\0': sb.append("\0\0\0"); break; case '\n': sb.append("\n\n\n"); break; case '\r': sb.append("\r\r\r"); break;</pre>

Figure 4.1: Bug Report (top) and developer patch (bottom) for bug 383 of the Closure Compiler

`com.google.javascript.jscomp.CodeGenerator` class. To find this method, based on the report, the developer can create test cases to expose the undesired behavior (e.g., returning a null value for “\0”). With these test cases and previously-created passing tests (of which Closure has 6,740), the developer could use existing spectrum-based fault localization formulae to generate a ranked list of methods. The list could then be inspected, in order, until the root cause of the bug is localized.

Generally, the ranked list contains the methods invoked by failing test cases. However, the Closure Compiler is a large project: This bug implicates 6,646 methods! Moreover, none of state-of-the-art SBFL formulae [121, 116, 118] localize the actual faulty method within the top 10 of the produced list. The two best-performing formulae for this defect (*GP13* and *GP19* proposed by Yoo et al. [121]) rank the first faulty method at position #64. Multric [118] assigns the highest suspiciousness score to the faulty method; however, there are more than 1000

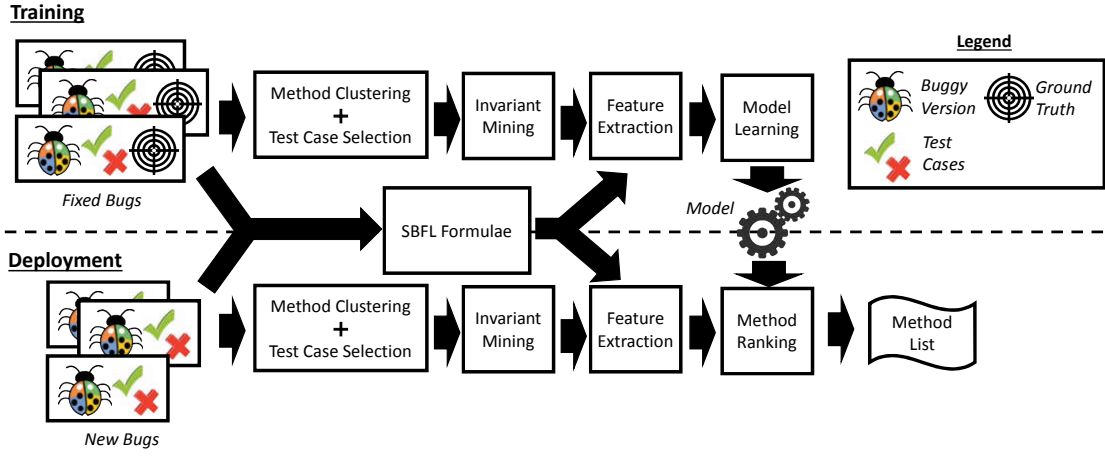


Figure 4.2: Overview of *Savant*'s Architecture

methods sharing this score. Existing SBFL approaches provide limited utility for the developer in this case.

On the other hand, *Savant* first uses Daikon to infer invariants from execution traces generated by both passing and failing test cases. Daikon [30]'s Invariant Diff utility on the inferred invariant sets implicates 556 suspicious methods with changes in learned invariants between passing versus failing execution types. Assuming any one of these methods is invoked by the failing test cases, I have already reduced the number of implicated methods from more than 6,000 to 556 regardless of SBFL rank. However, this is still an intractably large number. Therefore, I further rank the output using a learning to rank model built on historical changes in invariants of previously fixed bugs. The model assigns a score to each of the identified suspicious methods. I create a ranked list of suspicious methods sorted by the computed scores, and send to the developer for inspection.

Savant localizes the methods implicated in this example within the top-3 methods (Section 4.4), significantly outperforming the best SBFL approaches.

4.3 *Savant*

An overview of *Savant*'s architecture is shown in Figure 4.2. *Savant* works in two phases. In the training phase, *Savant* learns a statistical model that ranks methods

based on the likelihood that they are the root cause of a particular failure, based on features drawn from execution behavior on passing and failing test cases. This model is learned from training data consisting of a set of previously-repaired bugs, consisting of a buggy program version, passing and failing test cases, and ground truth bug locations. The training phase of *Savant* consists of four steps:

- *Method Clustering and Test Case Selection* (Section 4.3.1): The goal of this step is to limit the memory and runtime cost during invariant mining, while still enabling the inference of useful invariants. *Savant* first clusters methods executed by failing test cases. For every cluster, *Savant* selects a subset of particularly relevant passing and failing test cases. This step thus outputs a set of method clusters and their corresponding selected test cases.
- *Invariant Mining* (Section 4.3.2): *Savant* uses Daikon to record and infer method invariants for each cluster, based on program behavior on various sets of test cases. This step produces sets of program invariants, inferred from the execution of failing test cases, passing test cases, and their combination.
- *Feature Extraction* (Section 4.3.3): *Savant* uses the Daikon’s Invariant Diff utility to identify differences between method invariants inferred from the execution behavior on different sets of test cases. For instance, if invariant I holds over the set of passing test case executions, but not when the failing test cases are added, the methods where invariant I differs are *suspicious*. For methods whose invariant diffs are *non-empty*, *Savant* runs SBFL formulae to obtain suspiciousness scores. This produces a set of features per method for use as input to a model learning procedure, corresponding to either (1) the frequency of a type of invariant change for the method, or (2) a suspiciousness score computed by one of the SBFL formulae.

- *Model Learning* (Section 4.3.4): *Savant* takes the features and the ground truth locations to build a ranking *model*. This model is the overall output of the training phase that is passed to the deployment phase.

In the deployment phase (also discussed in Section 4.3.4), *Savant* takes as input a set of test cases (some failing, some passing), and a buggy program version and then uses the learned model to rank produce a ranked list of methods that are likely responsible for the failing test cases.

4.3.1 Method Clustering & Test Case Selection

Algorithm 2 Method Clustering & Test Case Selection

Require:

I : all methods executed by unsuccessful test cases
 P_i, F_i : passing and failing test cases, respectively
 M : maximum cluster size
 T : minimum acceptable coverage

Ensure: Clusters of methods and associated test suites

```

1: let  $C_s \leftarrow \text{reduce\_size}(kmeans(I, \frac{|I|}{M}))$ 
2: let  $C_r \leftarrow \emptyset$ 
3: for  $c_m \in C_s$  do
4:   let  $P_i^s \leftarrow \text{coverage\_sort}(P_i, c_m)$ 
5:   let  $P_c \leftarrow \emptyset$ 
6:   while  $\exists m \in c_m$  s.t.  $\text{coverage}(m, P_c) < M$  do
7:     let  $t \leftarrow \text{pop}(P_i^s)$ 
8:     if  $t$  covers at least one method  $m \in c$  then
9:        $P_c \leftarrow P_c \cup \{t\}$ 
10:    end if
11:  end while
12:   $C_r \leftarrow C_r \cup \{c_m, P_c\}$ 
13: end for
14: return  $C_r$ 

```

Savant must run the faulty program on passing and failing tests to record execution traces for invariant inference. Both trace collection and invariant mining can be very expensive, and the number of instrumented methods and executed test cases both contribute to the cost. For medium to large programs (e.g., Commons Math, Closure Compiler etc.), the runtime cost of running Daikon to infer

invariants for all methods executed by all test cases is very high (i.e., Daikon runs for many hours without producing results, or crashes with an out-of-memory exception). At the same time, I must collect sufficient data to support precise and useful invariant inference.

I resolve this tension via a set of novel heuristics for method clustering and test case selection for invariant inference on large programs for the purpose of fault localization. First, I exclude all methods not executed by any failing test cases. Second, I *cluster* methods by the passing test cases that execute them, and record execution traces for each cluster separately on all of the failing and then a selected subset of the passing test cases. Considering all passing test cases is unnecessarily costly, as many passing test cases are irrelevant to particular sets of methods.

Algorithm 2 describes my clustering and test case selection heuristic. This heuristic represents each method via a *coverage vector* describing all input passing test cases. The value of a dimension is one if the method is covered by the corresponding test case and zero otherwise. *Savant* uses k-means clustering [36] to group similar vectors (i.e., methods). K-means takes as input a set of vectors V and a fixed number of desired clusters k , and produces k clusters, each containing an arbitrary number of methods. I set k to $\lceil |I|/M \rceil$, where I is the number of methods to cluster, and M is the desired maximum cluster size. Because the resulting clusters may contain more than M methods, I heuristically split overlarge clusters into smaller groups no larger than size M . I keep selecting M methods randomly to create new groups until the cluster size is reduced to no more than M .

Savant then selects a subset of passing test cases for each cluster that covers its methods at least T times each. A perfect solution reduces to the NP-complete knapsack problem. I take a computationally simpler greedy approach, which suffices for my purpose. For each cluster, I sort passing test cases in descending

order by the number of methods in the cluster that each test covers. I then greedily take test cases from this list until all methods in the cluster are covered at least by at least T test cases. Both M and T can be tuned to fit experimental system capacity. Note that as the number of failing test cases is usually significantly less than the number of passing tests, I am still able to use all failing test cases of the faulty program for each cluster. The output of this step is the set of clusters C_r , each containing methods, where for each method cluster there is an associated subset P_c of passing test cases.

Both values of M and T should not be too large. If we set M and T to large values, there are less created clusters of methods and there are more test cases to execute in each cluster, which is very expensive. However, M and T should not be too small because that negatively affects quality of mined invariants. Furthermore, the tuning of M and T depends on the hardware capacity. In this work, I conducted the experiments on an Intel(R) Xeon E5-2667 system, and $M = 10$ and $T = 10$ is an appropriate choice to improve the speed of invariant inference. Note there are many optimal choices for M and T following the above discussion.

4.3.2 Invariant Mining

For each cluster produced in the preceding step, *Savant* traces execution information for each included method across all failing test cases as well as the selected passing test subset for that cluster. *Savant* collects execution information separately for the methods in the cluster using the following three sets of test cases: F_i , P_c , and $F_i \cup P_c$ of each method cluster c . For clarity, we refer subsequently to the execution of the failing test cases on a cluster c as F_c , but remind the reader that this involves running all failing test cases on each cluster. This information is then input to Daikon to infer invariants. We refer to the invariants inferred by Daikon as $\text{inv}(F_c)$, $\text{inv}(P_c)$, and $\text{inv}(F_c \cup P_c)$, respectively, and these sets of

invariants form the output of this step. *Savant* uses these sets to produce features for the learning-to-rank model construction, discussed next.

4.3.3 Feature Extraction

Savant extracts two different types of features: *invariant changes* features and *suspiciousness scores* features.

Invariant Change Features

For the first set of features, *Savant* uses Daikon’s Invariant Diff tool to describe changes in invariant sets between failing and passing program executions. In a nutshell, Invariant Diff recognizes changes in method invariants inferred from different sets of execution traces. The changes can consist of a transformation of an invariant into another (e.g., *OneOfString* to *SingleString*, or *NonZero* to *EqualZero*), or invariant removal or addition. My overall insight is that if an invariant I holds over successful test case executions (i.e., $I \in \text{inv}(P_c)$), but not when the failing test cases are added (i.e., $I \notin \text{inv}(F_c \cup P_c)$), the locations where invariant I differs are suspicious. These suspicious locations are at the entry and exit point of a method, suggesting the bug lies within that method. Thus, *Savant* ultimately analyzes and ranks only these suspicious methods, rather than all methods covered by the failing test cases.

Savant uses Invariant Diff to perform three types of comparisons per cluster: $\text{inv}(P_c) \times \text{inv}(F_c \cup P_c)$, $\text{inv}(F_c) \times \text{inv}(P_c)$, and $\text{inv}(F_c) \times \text{inv}(F_c \cup P_c)$. I refer to the output of Invariant Diff on these pairs as $\text{idiff}(P_c, F_c \cup P_c)$, $\text{idiff}(F_c, P_c)$, and $\text{idiff}(F_c, F_c \cup P_c)$, respectively. I then convert these invariant changes to features. Feature f of method m is a 3-tuple: $f = [I_A, I_B, \text{InvDiff}_{AB}]$. I_A is the type of the source invariant inferred from the execution of one set of test cases, I_B is the type of the target invariant to which I_A is transformed, and which holds in the execution of the other set of tests, and InvDiff_{AB} indicates the Invariant Diff’s

output type from which the change was discovered. I refer to I_A and I_B as the left-hand side (LHS) and right-hand side (RHS) invariant, or the *source* and *target* invariants, interchangeably. The value of feature f is then the frequency of the change between I_A and I_B in an InvDiff_{AB} comparison. Note that the values of I_A and I_B are the invariant types, rather than concrete invariants. Similarly, the value of InvDiff_{AB} in a feature’s 3-tuple is a label rather than the concrete value of Invariant Diff’s output. For example: **[OneOfString, SingleString, idiff(F_c, P_c)]** can be read as “A OneOfString invariant learned from execution of F_c becomes a SingleString invariant in the execution of P_c .”

Daikon supports many different types of invariants, including abstractions of concrete invariants. For example, `UnaryInvariant` abstracts the `LowerBound` and `NonZero` invariants. These invariant types create an inheritance hierarchy rooted at `daikon.inv.Invariant`². Thus, I enrich the feature set by replacing the LHS and RHS invariants of a feature with their abstract types to form a new feature. For example, **[UnaryInvariant, SingleString, idiff(F_c, P_c)]** abstracts **[OneOfString, SingleString, idiff(F_c, P_c)]**, and **[UnaryInvariant, SingleFloat, idiff($F_c, F_c \cup P_c$)]** abstracts **[LowerBoundFloat, SingleFloat, idiff($F_c, F_c \cup P_c$)]**.

Each such feature is a pair of invariants that reflects an abstraction of a method’s behavioral changes, and collecting many such features improves the chances that *Savant* successfully captures distinctive changes in the behavior of faulty methods. `Invariant` has 311 subclasses, and thus I have $311 \times 311 = 96,721$ potential invariant pairs (and thus overall features) across the three different Invariant Diff runs i.e., `idiff($P_c, F_c \cup P_c$)`, `idiff(F_c, P_c)`, and `idiff($F_c, F_c \cup P_c$)`.

Suspiciousness Scores Features

It is possible for methods to share similar changes to their invariants between sets of execution traces. These cases are more difficult for ranking models to

²<http://goo.gl/EPwNhV>

distinguish faulty and non-faulty methods. *Savant* therefore also includes suspiciousness scores output by spectrum-based fault localization tools as additional features. For each method implicated by changed invariants, *Savant* computes the suspiciousness scores output by 10 state-of-the-art spectrum-based fault localization formulae: ER1^a, ER1^b, ER5^a, ER5^b, ER5^c proposed by Xie et al. [116], GP02, and GP03, GP13, GP19 proposed by Yoo et al. [121] and Multric proposed by Xuan et al. [118]. I also include the suspiciousness scores output by the 25 SBFL formulae (including Ochiai [6]) used by Multric, resulting in 35 features extracted from suspiciousness scores. These features and invariant change features are then forwarded to the model learning and method ranking steps.

4.3.4 Model Learning and Method Ranking

Feature Normalization Before learning ranking models, I normalize feature values to a range of [0, 1], as follows:

$$v'_i = \begin{cases} 0 & \text{if } v_i < \min_i \\ \frac{v_i - \min_i}{\max_i - \min_i} & \text{if } \min_i \leq v_i \leq \max_i \\ 1 & \text{if } v_i > \max_i \end{cases} \quad (4.1)$$

where v_i and v'_i are the original and normalized values of the i^{th} feature of a suspicious method, \min_i and \max_i are the minimum and maximum values of the i^{th} feature inferred from the training data.

Model Learning and Method Ranking In the model learning step, *Savant* takes as input a set of fixed bugs, their corresponding features, and their ground truth faulty methods. The methods modified by the developers to fix the bugs provide this ground truth. For some bugs in the training set, the difference between the invariants of their faulty methods (for failed and passing test cases)

		Faulty Program	feature ₁	feature ₂	feature ₃	...	label
Method #1	1	$x_1^{(1,1)}$	$x_2^{(1,1)}$	$x_3^{(1,1)}$			$y^{(1,1)}$
Method #2	1	$x_1^{(1,2)}$	$x_2^{(1,2)}$	$x_3^{(1,2)}$...		$y^{(1,2)}$
			...				
Method #1	2	$x_1^{(2,1)}$	$x_2^{(2,1)}$	$x_3^{(2,1)}$			$y^{(2,1)}$
Method #2	2	$x_1^{(2,2)}$	$x_2^{(2,2)}$	$x_3^{(2,2)}$...		$y^{(2,2)}$
			...				
			...				

Figure 4.3: Input Data Format for Model Learning. $x_k^{(i,j)}$ corresponds to the value of feature k for method j in faulty program i . $y^{(i,j)}$ corresponds to the label (i.e., faulty or non-faulty) of method j for faulty program i .

can be empty. I exclude such bugs from the training set. Figure 4.3 shows the format of input data handled by the model learning step. Given input data in this format, I use rankSVM [53], an off-the-shelf learning-to-rank algorithm, to learn a statistical model that ranks methods based on such features.

In the method ranking step, my approach takes the features generated for a new bug as input to the learned model. Finally, *Savant* outputs a ranked list produced by the learned model to the developer for inspection.

4.4 Evaluation

4.4.1 Methodology

Dataset Many fault localization approaches [40, 7, 71, 118] are evaluated on artificial bugs (e.g., the SIR benchmark³, Steimann et al.’s benchmark [105]). However, it is unclear whether such bugs capture true characteristics of real bugs in real programs. Therefore, I evaluate *Savant* on 357 bugs from 5 different software projects in the Defects4J benchmark [41], a database of real, isolated, reproducible

³<http://sir.unl.edu/php/previewfiles.php>

Table 4.1: Dataset: The bolded components of names denote a shorthand abbreviation. “# Bugs” represents the number of bugs in each project. “Avg. KLOC”, “Avg. Tests”, and “Avg. Methods” correspond to average size of the program, number of test cases, and number of methods for each buggy version of each program, respectively.

Program	# Bugs	Average		
		KLOC	Tests	Methods
JFree Chart	26	132.9	1,824.9	7,782.5
Closure Compiler	133	345.6	7,200.1	7,479.5
Commons Math	106	111.8	2,905.0	4,792.3
Joda- Time	27	110.8	3,924.6	4,083.5
Commons Lang	65	52.6	1,859.0	2,151.1

software faults from real-world open-source Java projects intended to support controlled studies in software testing. The projects include a large number of test cases, and there exists at least one failing test case per bug. My choice of evaluation benchmark is inspired by influential previous work in the field [123, 59] that evaluates proposed fault localization approaches on real faulty programs. Table 4.1 describes the bugs and projects in the evaluation benchmark.

Comparative techniques I compare *Savant* against 11 state-of-the-art and well-known spectrum-based fault localization formulae described in Section 2.1.1 (i.e., ER1^a, ER1^b, ER5^a, ER5^b, ER5^c, GP02, GP03, GP13, GP19, Multric, and Ochiai) as well as Carrot [94]. I set the granularity of localized program entity to method, to match *Savant*. I extend Carrot to use all Daikon invariants and benefit from my test case selection strategy. This is important for scalability: without selection, Carrot takes hours to complete. The extended Carrot approach is referred to as Carrot⁺. In total, I compare *Savant* against 12 baselines.

Cross Validation I perform leave-one-out cross validation [35] across each of the five projects. Given a set of n bugs, I divide the set into n groups, of which $n - 1$ are used for training and the remaining serves as the test set. I repeat the

process n times, using a different group as the test set. I report *total* results across the n iterations. Compared to the standard 10-fold cross validation, leave-one-out cross validation is beneficial for evaluating on smaller datasets, as it provides more training data for each iteration, at the expense of training and evaluation time. *Savant* and Multric are the only two supervised learning techniques I evaluate, and thus I only perform cross validation for these two techniques, to mitigate the risk of overfitting. The other techniques are unsupervised.

Savant’s Settings For method clustering and test case selection, I set the maximum cluster size $M = 10$ and minimum acceptable size $T = 10$. *Savant* uses Daikon (version 5.2.8⁴) to infer invariants, scikit-learn⁵ 0.17.0 to perform k-means clustering, and rankSVM with linear kernel (version 1.95⁶) from LIB-SVM toolkit [17] for the learning to rank task with default settings. I perform all experiments on an Intel(R) Xeon E5-2667 2.9 GHz system with Linux 2.6.

Metrics I use three metrics to evaluate fault localization success:

- **acc@n** counts the number of successfully localized bugs within the top- n position of the resultant ranked lists. I use absolute ranks rather than percentages, following findings suggesting that programmers will only inspect the top few positions in a ranked list of potentially buggy statements [88]. I choose $n \in \{1, 3, 5\}$, computing $\text{acc}@1$, $\text{acc}@3$, and $\text{acc}@5$ scores. Note that if two program elements (i.e., methods) share the same suspiciousness score, I *randomly* break the tie. Higher is better for this metric.
- **wef@n** approximates the *wasted effort at n*, or effort wasted by a developer on non-faulty program elements before localizing the root cause a bug. $\text{wef}@n$ is calculated by the total number of non-faulty program elements

⁴<http://plse.cs.washington.edu/daikon/download/>

⁵<http://scikit-learn.org/>

⁶<https://goo.gl/pHku7x>

in top- n positions of ranked lists before reaching the first faulty program element, or the n^{th} program element in the ranked lists of all bugs. I again choose $n \in \{1, 3, 5\}$. Smaller is better.

- **Mean Average Precision (MAP)** evaluates ranking methods in information retrieval; I use it to evaluate the ranked list of suspicious elements produced by fault localization techniques. MAP is calculated using the mean of the *average precision* of all bugs, as follows:

$$AP = \sum_{i=1}^M \frac{P(i) \times pos(i)}{\text{number of faulty methods}}$$

where i is a rank of the method at the i^{th} position in the ranked list, M is total number of methods in the ranked list, and $pos(i)$ is a boolean function indicating whether the i^{th} method is faulty. $P(i)$ is the precision at i^{th} position starting from the beginning, defined as:

$$P(i) = \frac{\# \text{faulty methods in the top } i}{i}.$$

In cross-validation, I compute the Mean Average Precision (MAP) across all average precisions output across n iterations. Higher is better. Note that MAP is a very strict evaluation metric and its score is typically low (< 0.5) [101, 127]. For bugs appearing in single a method, even if all faulty methods appear in the top-3 position, the MAP score is only 0.33.

In spectrum-based fault localization, program elements are often assigned the same suspiciousness score. Thus, I repeat all metric calculations 100 times, using 100 different seeds to randomly break ties.

4.4.2 Research Questions

I investigate the following research questions:

RQ1: How effective is *Savant*? In this research question, we evaluate how effectively *Savant* identifies buggy methods for the 357 bugs, computing average $\text{acc}@n$, $\text{wef}@n$, and MAP scores ($n \in \{1, 3, 5\}$).

RQ2: How does *Savant* compare to previous approaches? In this research question, we compare *Savant* to 12 previous techniques across all evaluation metrics.

RQ3: What is the impact of the different feature sets on performance? By default, we use all features from both Invariant Diff and the suspiciousness scores. In this research question, we compare the two types of features to evaluate their individual contribution to *Savant*'s effectiveness by training and evaluating models on each set of features independently.

RQ4: How much training data does *Savant* need to work effectively? In the default setting, *Savant* uses $n-1$ out of n bugs as training data. We investigate the effectiveness of *Savant* with reduced amount of training data. We do this by evaluating *Savant* in a k-fold cross validation setting, for k ranging from 2–10.

RQ5: How efficient is *Savant*? In this research question, we measure the average running time needed for *Savant* to output a ranked list of methods for a given bug.

4.4.3 Findings

RQ1: Savant's Effectiveness.

In this research question, I evaluate how effectively *Savant* identifies buggy methods for the 357 bugs, computing average $\text{acc}@n$, $\text{wef}@n$, and MAP scores ($n \in \{1, 3, 5\}$). Table 4.2 shows the effectiveness of *Savant* on bugs from the five Defects4J projects. *Savant* successfully localizes 63.03, 101.72, and 122.00 out of 357 bugs in terms of average $\text{acc}@1$, $\text{acc}@3$, and $\text{acc}@5$ score, respectively. The wasted effort across all projects is 288.97, 811.14, and 1277.14 ($\text{wef}@1$, $\text{wef}@3$,

Table 4.2: *Savant*'s effectiveness in terms of average $acc@n$ ($n \in \{1, 3, 5\}$), $wef@n$ ($n \in \{1, 3, 5\}$) and MAP.

Project	Total Bugs	Avg. acc			Avg. wef			Avg. MAP
		@1	@3	@5	@1	@3	@5	
Chart	26	5.00	8.00	9.00	20.00	56.00	86.00	0.201
Closure	133	2.00	9.00	13.00	131.00	384.00	627.00	0.041
Math	106	22.03	36.72	47.00	82.97	226.14	348.14	0.261
Time	27	5.00	12.00	12.00	22.00	55.00	85.00	0.247
Lang	65	29.00	36.00	41.00	33.00	90.00	131.00	0.535
Overall	357	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

and $wef@5$, respectively). The overall average MAP score is 0.221. Among the five projects, *Savant* is most effective on Commons Lang, achieving the highest $acc@k$ (29, 36, and 41, respectively), and a MAP score of 0.535. Over these experiments, *Savant* terminated Daikon twice on a bug from the Math project due to the time limit. In total, I invoked Daikon 129,798 times for the 357 faulty versions.

RQ2: Savant vs. Previous work






In this research question, I compare *Savant* to 12 previous techniques across all evaluation metrics. Table 4.3, 4.4, and 4.5 show the effectiveness of the 12 baseline approaches on my dataset. Among the baselines, the top four performers are $ER1^b$, $GP13$, $GP19$ and Multric, and they achieve more or less the same score for many metrics. The absolute best performers are $ER1^b$ and $GP13$.

Figure 4.4 and 4.5 demonstrate the comparison between *Savant* and all baselines in $acc@N$, $wef@N$, and MAP ($N \in \{1, 5, 10\}$). According to the figures, *Savant* outperforms these baselines in all metrics. It outperforms $ER1^b$ and $GP13$ by 57.73%, 56.69%, and 43.13% in terms of average $acc@1$, $acc@3$, and $acc@5$ scores. The wasted effort of *Savant* is lower than those of $ER1^b$ and $GP13$ by 8.85%, 10.94%, and 12.78% ($wef@1$, $wef@3$, and $wef@5$ respectively). In terms of average MAP score, my approach is more effective than $ER1^b$ and $GP13$ by

Table 4.3: Effectiveness of Baseline Approaches (Part I).

Bug	Project	Average acc			Average wef			MAP
		@1	@3	@5	@1	@3	@5	
<i>ER1^a</i>	Chart	0.93	2.42	3.50	25.07	72.83	118.27	0.08
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.31	19.70	28.80	97.69	275.74	434.00	0.16
	Time	1.00	4.48	7.00	26.00	72.52	114.00	0.07
	Lang	2.46	17.86	25.54	62.54	164.62	246.95	0.18
	Overall	15.31	50.61	74.10	341.69	971.06	1547.44	0.11
<i>ER1^b</i>	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	Overall	39.96	64.92	85.24	317.04	911.42	1464.26	0.15
<i>ER5^a</i>	Chart	4.00	4.49	5.56	22.00	65.44	107.00	0.15
	Closure	0.31	0.69	1.17	132.69	397.48	661.40	0.01
	Math	2.41	7.32	11.68	103.59	303.63	494.48	0.06
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	23.20	31.83	38.32	41.80	113.22	169.24	0.38
	Overall	32.92	47.33	59.73	324.08	951.77	1552.12	0.10
<i>ER5^b</i> <i>ER5^c</i>	Chart	0.97	2.95	4.64	25.03	72.04	115.56	0.07
	Closure	0.31	0.69	1.17	132.69	397.48	661.40	0.01
	Math	2.36	7.21	11.54	103.64	303.87	494.99	0.06
	Time	0.25	0.85	1.37	26.75	79.40	130.92	0.02
	Lang	5.37	15.60	24.27	59.63	163.13	248.92	0.17
	Overall	9.26	27.30	42.99	347.74	1015.92	1651.79	0.06

Savant's improvement

Legend	 improvement <10%	 10% ≤ improvement <20%
	 20% ≤ improvement < 50%	 50% ≤ improvement <100
	 100% ≤ improvement	

51.37%. Overall, *Savant* outperforms all popular and state-of-the-art baseline approaches across all measured metrics.

Note that although *Multric* includes *Ochiai* as a feature, it does not considerably outperforms *Ochiai*. *Multric* outperforms *Ochiai* by 6.73% and 0.09% in terms of average acc@1 and acc@3 scores, and the wasted effort of *Multric* is only lower than that of *Ochiai* by 0.77% and 0.72% in terms of average wef@1 and wef@3 score. For the other metrics (average acc@10, wef@10 and MAP scores), *Ochiai* outperforms *Multric*. This result is in contrast with that of *Savant* which

Table 4.4: Effectiveness of Baseline Approaches (Part II).

Bug	Project	Average acc			Average wef			MAP
		@1	@3	@5	@1	@3	@5	
GP02	Chart	0.00	0.00	0.00	26.00	78.00	130.00	0.01
	Closure	0.10	0.30	0.50	132.90	398.36	663.49	0.00
	Math	0.15	0.47	0.83	105.85	317.01	527.50	0.01
	Time	0.00	0.00	0.00	27.00	81.00	135.00	0.00
	Lang	8.00	11.00	12.12	57.00	167.00	273.78	0.12
	Overall		8.25	11.77	13.45	348.75	1041.37	1729.77
GP03	Chart	0.00	1.00	2.00	26.00	76.00	125.00	0.02
	Closure	0.65	1.01	1.02	132.35	396.37	660.34	0.01
	Math	0.31	0.90	1.37	105.69	316.14	525.60	0.01
	Time	0.00	0.00	0.00	27.00	81.00	135.00	0.01
	Lang	8.00	13.00	16.00	57.00	163.00	261.00	0.14
	Overall		8.96	15.91	20.39	348.04	1032.51	1706.94
GP13	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	Overall		39.96	64.92	85.24	317.04	911.42	1464.26
GP19	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.60	6.11	9.22	130.40	385.42	634.37	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	Overall		39.95	64.88	85.20	317.05	911.49	1464.41

Savant's improvement

Legend	 improvement <10%	 10% ≤ improvement <20%
	 20% ≤ improvement < 50%	 50% ≤ improvement <100
	 100% ≤ improvement	

outperforms all existing techniques by a much larger margin (e.g., 57.73% versus 6.73% for acc@1).

RQ3: Different Sets of Features

By default, I use all features from both Invariant Diff and the suspiciousness scores. In this research question, I compare the two types of features to evaluate their individual contribution to *Savant's* effectiveness by training and evaluating models on each set of features independently. Table 4.6 shows the effectiveness of

Table 4.5: Effectiveness of Baseline Approaches (Part III).

Bug	Project	Average acc			Average wef			MAP
		@1	@3	@5	@1	@3	@5	
Ochiai	Chart	2.00	4.00	6.00	24.00	69.00	109.00	0.13
	Closure	1.95	4.70	8.77	131.05	388.77	639.29	0.04
	Math	8.27	19.71	28.82	97.73	275.71	433.99	0.16
	Time	5.50	7.00	9.00	21.50	62.50	99.50	0.12
	Lang	18.84	28.50	35.09	46.16	125.47	187.35	0.34
	OA	36.56	63.91	87.68	320.44	921.45	1469.13	0.14
Multric	Chart	4.35	6.44	8.18	21.65	62.14	98.35	0.15
	Closure	1.83	5.14	7.84	131.17	388.24	639.73	0.03
	Math	6.33	16.88	25.36	99.67	283.34	448.52	0.14
	Time	3.71	5.47	6.52	23.29	67.33	108.65	0.10
	Lang	22.80	30.04	34.06	42.20	113.80	177.57	0.36
	OA	39.02	63.97	81.96	317.98	914.85	1472.82	0.14
Carot+	Chart	2.05	3.86	5.16	22.95	65.95	104.24	0.11
	Closure	0.35	1.01	1.88	132.65	396.99	659.67	0.01
	Math	5.81	13.84	20.48	99.19	284.99	457.32	0.10
	Time	0.90	1.86	2.36	26.10	76.69	126.25	0.06
	Lang	20.48	33.84	39.58	41.52	102.21	146.49	0.41
	OA	29.59	54.41	69.46	322.41	926.83	1493.97	0.12

Savant’s improvement






Legend	 improvement <10%	 10% ≤ improvement <20%
	 20% ≤ improvement < 50%	 50% ≤ improvement <100
	 100% ≤ improvement	

Table 4.6: Savant’s effectiveness using different features.

Feature Set	Avg. acc			Avg. wef			Avg. MAP
	@1	@3	@5	@1	@3	@5	
Inv. Changes	55.23	83.50	105.00	296.77	848.64	1346.14	0.179
Susp. Scores	53.81	86.56	105.09	298.19	845.11	1340.82	0.214
Default	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

Savant using invariant change features, suspiciousness scores features, and their combination (the default). Savant is less effective if only one type of features is used to construct ranking models, across all metrics. Savant using only suspiciousness scores is more accurate than using only invariant changes according to most metrics, though notably *not* acc@1, nor wef@1. Regardless of the features used in the model, and unlike the baselines, Savant only ranks methods that have

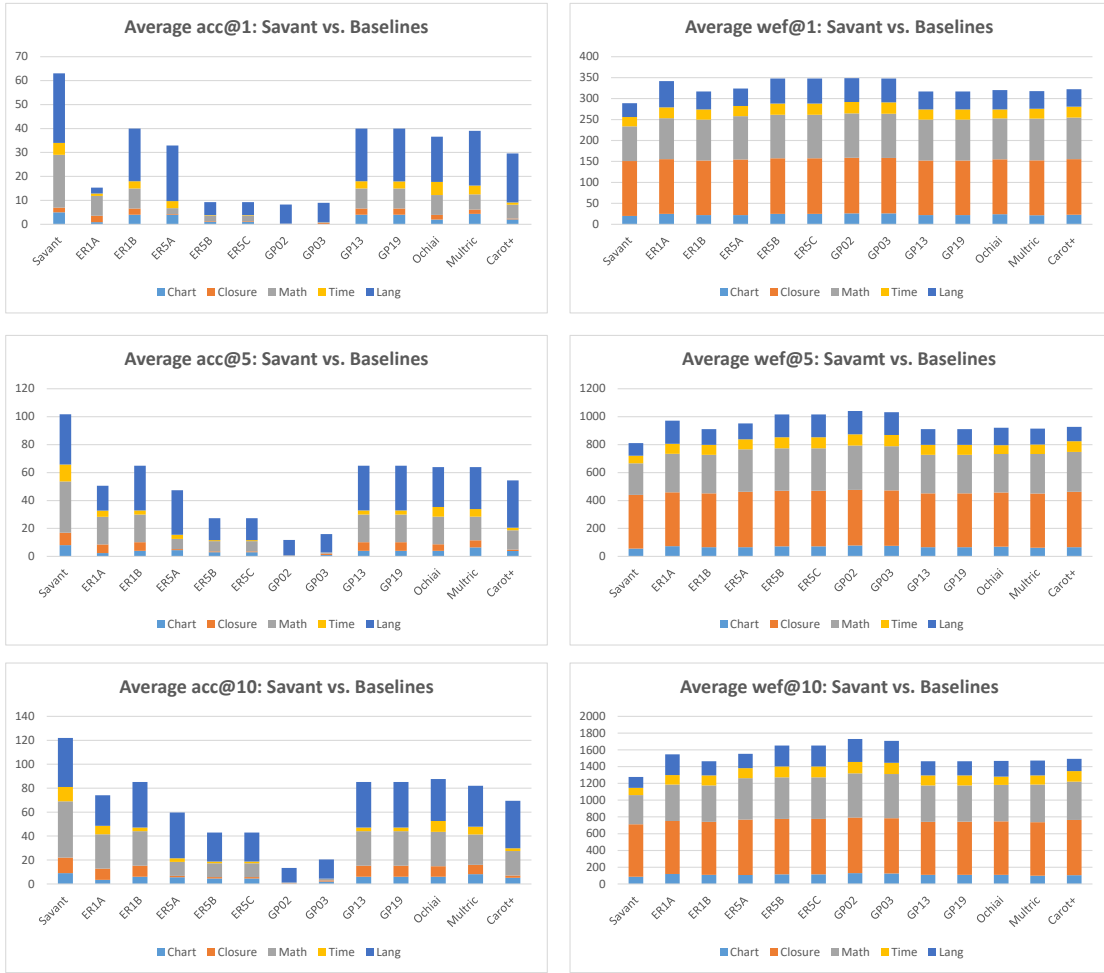


Figure 4.4: Top N ($N \in \{1, 5, 10\}$): Savant vs. Baselines in Bar Charts.

changes in invariants in the two set of execution traces, instead of all methods in faulty programs. This explains why *Savant* built using only suspiciousness score features outperforms Multirc. Overall, the combination of the two feature types significantly improves the effectiveness of *Savant*.

RQ4: Varying Training Data Size

In the default setting, *Savant* uses $n-1$ out of n bugs as training data. I investigate the effectiveness of *Savant* with reduced amount of training data. I do this by evaluating *Savant* in a k-fold cross validation setting, for k ranging from 2–10. Table 4.7 shows the effectiveness of *Savant* in various cross validation settings.

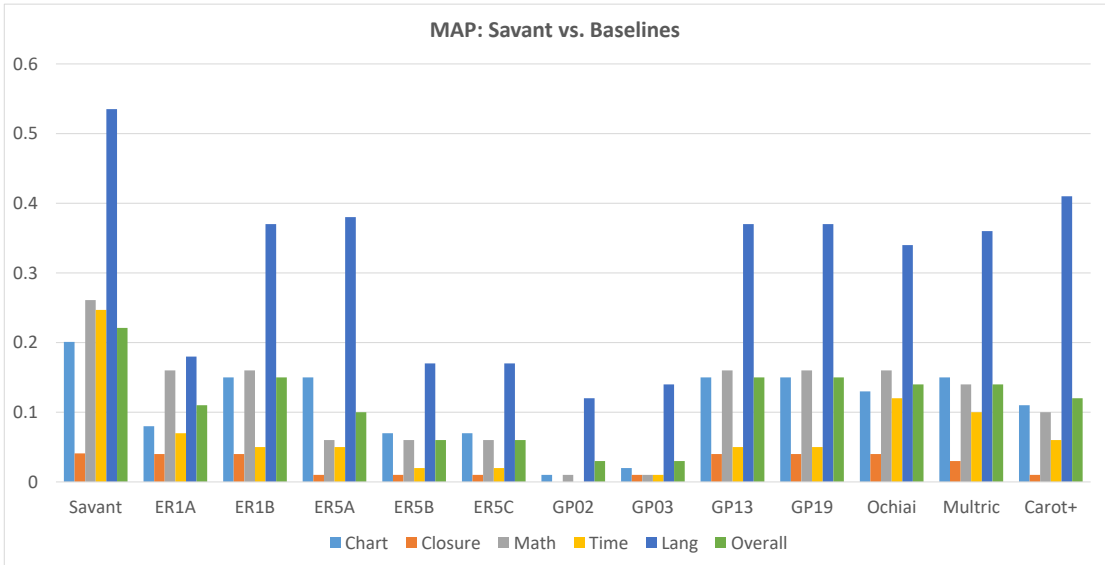


Figure 4.5: MAP: Savant vs. Baselines in Bar Charts.

Table 4.7: Varying training data size: average acc@n ($n \in \{1, 3, 5\}$), wef@n ($n \in \{1, 3, 5\}$) and Mean Average Precision (MAP). “K” represents the number of folds in cross-validation setting. “Default” stands for leave-one-out cross-validation.

K	Avg. acc			Avg. wef			Avg. MAP
	@1	@3	@5	@1	@3	@5	
10	61.53	94.72	118.50	290.47	825.64	1298.64	0.215
9	53.03	90.72	110.50	298.97	844.14	1335.14	0.204
8	63.85	94.98	109.69	288.15	815.86	1302.45	0.219
7	68.35	102.19	111.50	283.65	797.83	1276.33	0.220
6	59.35	96.48	117.69	292.65	818.36	1294.45	0.222
5	68.85	94.50	112.00	283.15	809.02	1292.52	0.223
4	63.53	92.22	110.05	288.47	817.14	1307.59	0.219
3	62.50	93.00	114.46	289.50	822.50	1307.54	0.216
2	59.68	90.07	110.00	292.32	825.27	1314.61	0.211
Default	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

Although the effectiveness of *Savant* varies from setting to setting, the range of effectiveness is fairly small. Among the settings, 5-fold cross validation ($k = 5$) achieves the best performance in most metrics compared to others, but there is no setting that outperforms the others across all metrics. I conclude that the amount of training data has little impact on *Savant*’s accuracy.

Table 4.8: Running time of *Savant* (in tens of seconds)

Project	Mean	Standard Deviation
JFreeChart	15.30	2.85
Closure Compiler	318.45	39.71
Commons Math	41.01	6.09
Joda-Time	23.78	4.59
Commons Lang	19.70	3.41
Overall	138.94	142.32

RQ5: Efficiency

In this research question, I measure the average running time needed for *Savant* to output a ranked list of methods for a given bug. Table 4.8 shows average running time for *Savant* on my dataset, including both learning and ranking. The average time to output a ranked list of methods for a given bug from any of the five projects is 138.94 tens of seconds, with a standard deviation of 142.32 tens of seconds. This average is dominated by the running time on a single project (the Closure Compiler); the median running time is 23.78 tens of seconds (Joda-Time). Among the projects, *Savant* has lowest average execution time on JFreeChart bugs (15.30 tens of seconds). Closure Compiler bugs take the longest to localize. The Closure Compiler is considerably larger than the other projects, leading to a longer running time in constructing ranking models. I observe, however, that this running time is conservative, since it includes the training phase, which could be amortized across different bug localization efforts, and overall is reasonable in practice.

4.5 Conclusion

I have evaluated my solution on a set of 357 bugs from 5 programs in the Defects4J benchmark. My evaluation demonstrates that *Savant* can successfully localize 63.03, 101.72, 122 bugs on average within the top 1, top 3, and top 5 listed methods, respectively. I have compared *Savant* against 10 SBFL techniques that

have been proven to outperform many other SBFL techniques, a hybrid SBFL technique that also uses learning-to-rank (Multric), and an extended version of an SBFL technique that also uses likely invariants (Carrot⁺). *Savant* can locate 57.73%, 56.69%, and 43.13% more bugs at top 1, top 3, and top 5 methods as compared to the best performing baseline techniques.

In the future work, I plan to improve *Savant* further by selectively including a subset of invariants specialized for a target buggy program version and its spectra. I also plan to include a refinement process which incrementally adds or removes invariants to produce a better ranked list of methods. Furthermore, I plan to extend my evaluation to include more bugs beyond those in the Defects4J benchmark and compare *Savant* against other fault localization approaches. I also plan to investigate the impact of number of passed and failed test cases as well as other factors on the effectiveness of *Savant*.

Chapter 5

SYNERGIZING SPECIFICATION MINERS THROUGH MODEL FISSIONS AND FUSIONS

Software systems are often developed and released without formal specifications. For those systems that are formally specified, developers have to continuously maintain and update the specifications or have them fall out of date. To deal with the absence of formal specifications, researchers have proposed techniques to infer the missing specifications of an implementation in a variety of forms, such as finite state automaton (FSA). Despite the progress in this area, the efficacy of the proposed specification miners needs to improve if these miners are to be adopted. In this work, I propose *SpecForge*, a new specification mining approach that synergizes many existing specification miners. SpecForge decomposes FSAs that are inferred by existing miners into simple constraints, through a process I refer to as model fission. It then filters the outlier constraints and fuses the constraints back together into a single FSA (i.e., model fusion).

5.1 Introduction

The short time-to-market and rapid evolution of software has led to software systems and libraries that are released without any documented specifications. Even when a system includes formal specifications, these specifications may become quickly out of date as the software evolves [126]. Finally, developers often lack the necessary skill and motivation to write formal specifications, as this takes significant time and manual effort [43]. The unavailability of specifications negatively impacts the maintainability and reliability of systems. Without specifications developers find code comprehension more difficult, and software becomes more error-prone as bugs are introduced due to mistaken assumptions. Further-

more, without a formal specification, developers cannot take advantage of some state-of-the-art bug finding and testing tools that require formal specifications as an input [19, 78].

In this work we propose **SpecForge**, an automated approach to synergize the many existing FSA-based specification mining algorithms. SpecForge first uses existing specification miners to infer a set of FSAs. It then uses these to generate a superior FSA. SpecForge first performs *model fissions* to extract important constraints that are common across the mined FSAs. SpecForge then performs *model fusions* to combine the extracted constraints into one FSA model. Both model fission and model fusion processes are completely automated. In this work, we use a set of 6 constraint templates to generate constraints, some of which were proposed by Dwyer et al. [29] and Beschastnikh et al. [14]. SpecForge checks whether one or more instances of these constraint templates are observed in a mined model. Constraints corresponding to models generated by various specification miners are then merged together while the outlier constraints are identified and omitted.

5.2 SpecForge

5.2.1 Overall Architecture

Figure 5.1 illustrates the architecture of SpecForge. SpecForge takes as input a set of execution traces of an API and outputs a finite state automaton (FSA). SpecForge has three steps: (1) model construction, (2) model fission, and (3) model fusion.

In the model construction step, the input traces are fed as inputs to N different FSA-based specification miners. Each miner infers a FSA according to its underlying mining algorithm: FSA_1, \dots, FSA_N . Many different specification mining algorithms have been proposed in the literature and in this work I focus on the $N = 7$ algorithms proposed by Krka et al. [46] (i.e., Traditional 1-tails, Traditional 2-tails, CONTRACTOR++, SEKT 1-tails, SEKT 2-tails, Optimistic

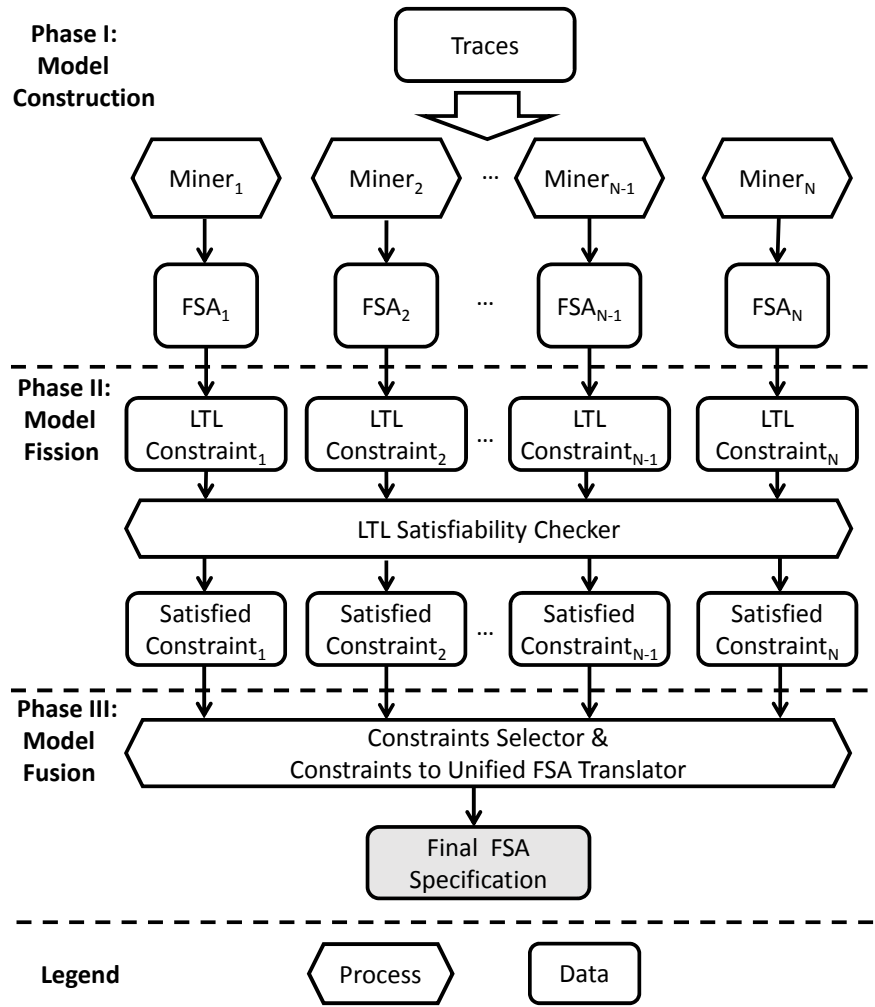


Figure 5.1: SpecForge Overview

TEMI, and Pessimistic TEMI), which are described in Section 2.2.1.

Once the specification miners infer their respective FSAs, SpecForge unifies these FSAs into one model. First, each inferred FSA is deconstructed into a set of constraints (*model fission*). Based on some criteria, the *strongly supported* constraints are selected from this set. Finally, the selected constraints are fused to form the final specification (*model fusion*). In the next two sections I further describe the model fission and model fusion steps.

5.2.2 Model Fission

The goal of this phase is to break a single FSA (e.g., FSA_i) into a set of basic building blocks that can be compared to blocks from other FSAs and used to build new FSAs. A key observation in my work is that a FSA can be thought of as a collection of *ordering constraints* among events. These ordering constraints can be shared by more than one FSA and are suitable building blocks for other FSAs. A classic formalism for specifying ordering constraints is Linear Temporal Logic (LTL) [90]. I use LTL in this work to specify ordering constraints between events.

The model fission process consists of two steps: constraint enumeration and constraint checking. In the first step, I generate a set of LTL constraint that may or may not be satisfied by the FSA. In the constraint checking step I filter out those LTL constraints that are not satisfied by the FSA.

Constraint Enumeration: It is impossible to check all possible LTL constraints. I therefore consider just the LTL constraints that fit the following six templates, each of which relates two events:

- a is **always followed by** b (denoted by $AF(a, b)$): an occurrence of event a must be *eventually* followed by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow XF b)$.
- a is **never followed by** b (denoted by $NF(a, b)$): there are *no* occurrences of event b after an occurrence of event a in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow XG(\neg b))$.
- a is **always preceded by** b (denoted by $AP(a, b)$): an occurrence of event a must be preceded by event b in the execution trace. In LTL, this rule is expressed as: $\neg a W b$.
- a is **always immediately followed by** b (denoted by $AIF(a, b)$): an oc-

currence of event a must be *immediately* followed by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow X b)$.

- a is **never immediately followed by b** (denoted by $\text{NIF}(a, b)$): there are no occurrences of event b immediately after any occurrence of event a in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow X(\neg b))$.
- a is **always immediately preceded by b** (denoted by $\text{AIP}(a, b)$): an occurrence of event a must be *immediately* preceded by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $F(a) \rightarrow (\neg a U(b \wedge Xa))$

Two of the six templates (i.e., always followed by, and always preceded by) correspond to two of the most commonly used LTL constraints (i.e., response and precedence) based on the survey by Dwyer et al. [29]. Another two templates (i.e., never followed by, and never immediately followed by) were introduced by Beschastnikh et al. [14] and have been demonstrated to be useful for describing FSA mining algorithms. The last two templates (i.e., always immediately followed by, and always immediately precedes) are newly introduced in this work. As a result, the bottom three templates are variations of the first three templates with the additional “immediately” requirement.

Given a set of execution traces, SpecForge enumerates all possible event pairs that appear in the traces. For each pair of events, e.g., a and b , I construct six possible LTL constraints corresponding to $\text{AF}(a, b)$, $\text{NF}(a, b)$, $\text{AP}(a, b)$, $\text{AIF}(a, b)$, $\text{NIF}(a, b)$, and $\text{AIP}(a, b)$. These constraints form the input to the constraint checking step.

Constraint Checking: This step checks the satisfiability of each of the generated LTL constraints in the enumeration step in the FSA model. For this checking I use the SPIN model checker [38], converting the FSA model into SPIN’s Promela language. This process filters out those LTL constraints that are not satisfied

by the FSA. At the end of this step the FSA is decomposed into a set of LTL constraints based on the six templates listed above; each constraint is satisfied by the FSA.

5.2.3 Model Fusion

The model fusion phase in SpecForge takes as input the sets of LTL constraints for each of the inferred FSAs. For the inferred FSAs FSA_1, \dots, FSA_N , I denote the corresponding sets of LTL constraints as C_1, \dots, C_N . That is, C_i is a set of constraints $\{C_{i1}, \dots, C_{iM_i}\}$ such that C_{ij} is based on one of the templates above and is satisfied by FSA_i .

The fusion process first selects LTL constraints from these input sets and then fuses the selected constraints into a new FSA. The fusion process contains the following three steps: (1) constraint selection, (2) constraint to model translation, and (3) unified model construction. I described each of these steps below.

Constraint Selection: The goal of this step is to select a sub-set of LTL constraints from the sets of all input constraints. In this work, I consider the following four heuristics for selecting constraints:

- **Union:** This heuristic assumes that all of the generated constraints are correct and any one set is incomplete. It returns the union of all the constraint sets: $\cup_{1 \leq i \leq N} C_i$.
- **Majority:** Unlike the Union heuristic this heuristics assumes that some of the constraints are incorrect, but it assumes that those constraints that are in common across a majority of the constraint sets are correct. This heuristic returns the union of all constraints that are satisfied by the majority of the FSAs. Let $\text{num-containing}(C_{ij})$ be the number of input constraint sets containing C_{ij} . This heuristic returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) \geq N/2\}$.

- **Satisfied By $\geq x$:** This heuristic generalizes the above heuristics. I deem a constraint as correct if it is satisfied by at least x FSAs. For $x > N/2$ this heuristic is at least as strict as the Majority heuristic. Otherwise, it is more lenient. This heuristic returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) \geq x\}$.
- **Intersection:** The final heuristic is the most conservative. It assumes that a correct constraint must have been satisfied by all inferred FSAs. It returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) = N\}$.

Constraint to Model Translation: At the end of the previous step I have a set of selected constraints. In this step, I convert each constraint into a simple FSA (see Figure 5.3). Each simple FSA involves two distinct events in a given alphabet (e.g., a and b). Note that in Figure 5.3 not all rejecting states are shown for each FSA.

For example, Figure 5.3 (a) represents the FSA corresponding to the LTL constraint $AF(a, b)$. In Figure 5.3 (a), accepting state is represented by a double circle and rejecting state is represented by a single circle. The initial state is $S1$ and whenever the event a happens, state $S2$ is entered. Next, whenever event b happens from state $S2$, state $S1$ is entered again. For example, this FSA accepts the sentence $aabab$.

In addition to the six simple FSAs in Figure 5.3, I consider one special FSA which describes the rule “ a is never immediately followed by a ” (the two events are the same event). In this case, I construct a FSA (see Figure 5.2) which is slightly different from Figure 5.3 (d).

Unified Model Construction: In this step, SpecForge combines the constraint FSA models generated in the previous step into a unified FSA. Each model specifies a language or a set of execution traces that it accepts. I want the unified FSA to accept an intersection of these languages (i.e., a set of sentences in which each is accepted by *all* of the simple models). To construct such a unified FSA,

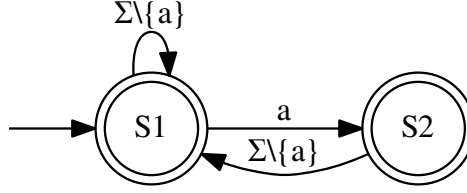


Figure 5.2: FSA for “ a is never immediately followed by a ”.

Table 5.1: List of Target Library Classes and Analyzed Programs.

Target Library Classes		Client Programs
Full Name	Short Name	
java.util.ArrayList	ArrayList	Dacapo fop
java.util.HashMap	HashMap	Dacapo h2
java.util.HashSet	HashSet	Dacapo h2
java.util.Hashtable	Hashtable	Dacapo xalan
java.util.LinkedList	LinkedList	Dacapo avrora
java.util.StringTokenizer	StringTokenizer	Dacapo batik
org.apache.xalan.templates.ElemNumber\$NumberFormatStringTokenizer	NFST	Dacapo xalan
DataStructures.StackAr	StackAr	StackArTester
java.security.Signature	Signature	Columba, jFTP
org.apache.xml.serializer.ToHTMLStream	ToHTMLStream	Dacapo xalan
java.util.zip.ZipOutputStream	ZipOutputSt	JarInstaller
org.columba.ristretto.smtp.SMTPProtocol	SMTPProtocol	Columba
java.net.Socket	Socket	Voldemort

SpecForge performs intersection over the FSAs corresponding to the selected constraints using the `dk.bricks.automaton` library [81]. The unified FSA will always be a connected FSA since it is always possible to represent a set of sentences as one connected FSA.

5.3 Evaluation

In this section, I first describe my methodology in evaluating SpecForge against a number of baselines. I then describe four research questions and present my experimental results that answer these questions. I finish the section by discussing

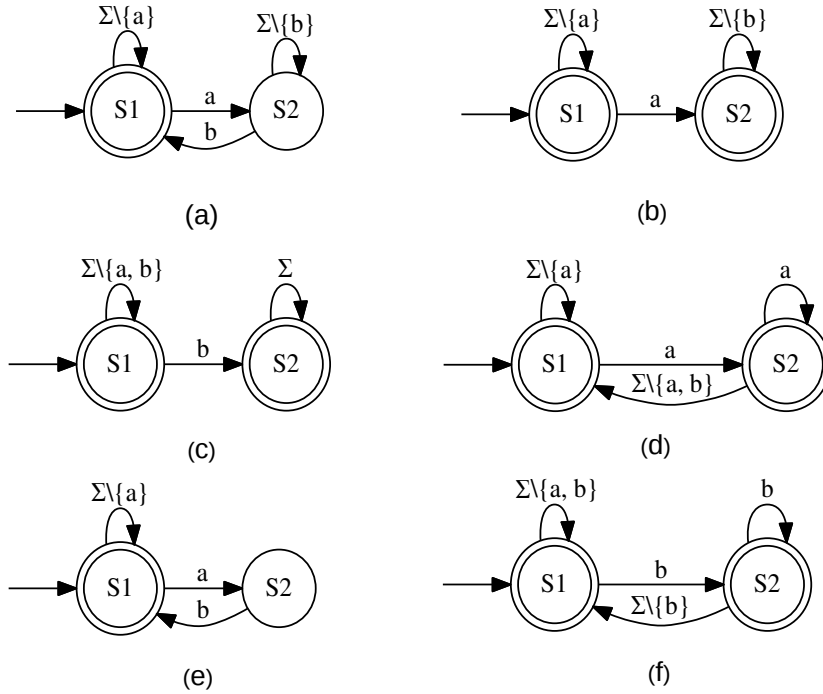


Figure 5.3: Translations of LTL expressions to FSAs: (a) a is always followed by b , (b) a is never followed by b , (c) a is always preceded by b , (d) a is never immediately followed by b , (e) a is always immediately followed by b , (f) a is always immediately preceded by b .

remaining, untapped potentials, of my approach.

5.3.1 Methodology

Target Library Classes: In this work, I evaluate the effectiveness of the SpecForge specification miner in generating behavioral models of 13 library classes. The list of 13 library classes is listed in Table 5.1. The 7 underlying specification miners on top of which SpecForge is built require a set of execution traces to infer FSAs. These traces were obtained by running a set of test cases. I use traces from passing test cases as they are likely to capture correct program behaviour. I make use of a number of execution traces made available by Krka et al. and generate additional execution traces by running programs in the DaCapo benchmark. The list of client programs that had been run to generate the traces are also listed in

Table 5.1.

Ground Truth Models: To evaluate the quality of a generated FSA, I need a ground truth FSA. My ground truth FSAs are taken from those that were created by Krka et al. [46] and Pradel et al. [93]. Following Krka et al., I remove edges and nodes from the FSAs that do not appear in the execution traces that I use to mine the model. Also, since the models were not created by library creators, to ensure correct ground truths, I check the ground truth models against documented specifications of library usage. I corrected a few errors in the ground truth FSAs that were manually created by Krka et al. and Pradel et al. that do not follow the documented specifications. I exclude `net.sf.jftp.net.wrappers.SftpConnection` from my experiments (which was considered by Krka et al.) due to the lack of documentation, which prevented us from verifying its ground truth model. I exclude some library classes whose models are made available by Pradel et al. due to difficulties in running Daikon to collect execution traces from some of the JDK libraries¹. The corrected ground-truth models are publicly available: <https://github.com/ModelInference/SpecForge>

Evaluation Metrics: To measure the effectiveness of SpecForge, I use precision and recall introduced by Lo and Khoo [26]. These have been previously used to evaluate many different specification mining algorithms, e.g., [46, 26]. Precision and recall are computed by comparing the language that is accepted by an inferred FSA with the language that is accepted by a ground truth FSA. Precision refers to the proportion of sentences that are accepted by the inferred model that is also accepted by the ground truth model. Recall refers to the proportion of sentences that are accepted by the ground truth model that is also accepted by the inferred model. I also compute F-measure, which is the harmonic mean of precision and

¹I have checked with Daikon developers who responded: “there is not an easy way to generate invariants within the JDK; I assume that the libraries are correct” [97].

recall and is defined as:

$$F\text{-measure} = 2 \times \frac{\textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}}$$

There is often a tradeoff between precision and recall: one can gain precision by sacrificing recall (and vice versa). For example, SpecForge I omit more constraints in the model fusion phase (resulting in a more general model that accepts more traces) to achieve higher recall. However, this *may* reduce precision (if correct constraints are removed in the process)². In the extreme, if only one *correct* rule is selected, the precision will be 100%, but recall will be low. F-measure is often used as a summary measure that evaluates if an increase in precision outweighs a reduction in recall (and vice versa). Thus, I use F-measure as a final yardstick to evaluate the effectiveness of specification mining algorithms.

In the evaluation of precision and recall, I need to generate a set of sentences that characterize the language that is accepted by a FSA. To generate sufficient number of sentences that characterize the language that is accepted by an FSA, I follow the procedure described by Lo et al. in their recent empirical study paper that compares the effectiveness of various existing specification mining algorithms [67]. I generate a set of sentences that are accepted by the FSA such that each edge of the FSA is covered at least 10 times (10-transition-coverage). Since some of the inferred automata is an NFA (Non-deterministic Finite Automata) and are very large, to keep the number and length of traces manageable, I limit the number of traces to 10,000 and the length of traces to 100.

Default Configuration: My proposed approach has a number of parameters that can be tuned. The first parameter is the selection of constraint templates that are used in the constraint enumeration step. The second parameter is the selection of heuristic used in the constraint selection step. By default, I make use

²If only bad constraints are removed, then precision will also improve.

of *all* the six constraint templates for the constraint enumeration step, and the Intersection heuristic for the constraint selection step.

5.3.2 Research Questions

RQ1: How effective is the SpecForge specification mining approach?

The effectiveness of a specification mining approach affects the usefulness of the mined specification for program comprehension and for automated program analysis. To answer this research questions, we measure the precision, recall, and F-measure of SpecForge in inferring behavioral models of the 13 library classes described in Section 5.3.1.

RQ2: How much does SpecForge improve over existing specification mining approaches?

Many specification mining approaches that analyze execution traces and output a finite state automaton have been proposed in the literature. SpecForge is built on top of 7 existing approaches. To answer this research question, we compare the precision, recall, and F-measure of SpecForge with those of the 7 existing approaches for each of the 13 library classes that we investigate in this work.

RQ3: What is the impact of changing the constraint templates used in the constraint enumeration step?

In this paper, we consider six different constraint templates. Some constraints may capture important properties in a FSA that a miner successfully “generalizes” from a set of execution traces, while others may capture the idiosyncrasies of a FSA miner which “overfit” the execution traces. In this research question, we investigate if some constraint templates are prone to capturing incorrect constraints. To answer this question, we evaluate the effectiveness of SpecForge when only some of the constraint templates are considered. We then highlight the effect of adding and omitting some constraint templates.

Table 5.2: Precision, Recall, and F-measure: SpecForge with Default Configuration.

Target Library Classes	Precision	Recall	F-measure
ArrayList	100.00%	65.08%	78.85%
HashMap	100.00%	44.02%	61.13%
HashSet	100.00%	55.44%	71.33%
Hashtable	100.00%	44.11%	61.22%
LinkedList	100.00%	82.80%	90.59%
StringTokenizer	60.00%	74.15%	66.33%
NFST	92.00%	30.63%	45.96%
SMTPProtocol	93.73%	45.00%	60.81%
Signature	100.00%	24.32%	39.13%
Socket	77.07%	40.86%	53.41%
StackAr	54.62%	100.00%	70.65%
ToHTMLStream	100.00%	60.00%	75.00%
ZipOutputStream	100.00%	43.18%	60.32%
Average	90.57%	54.58%	64.21%

RQ4: What is the impact of changing the heuristics used in the constraint selection step?

In this work, we propose a number of heuristics that we can adopt when selecting constraints that have been extracted from the input FSAs. These heuristics, presented in Section 5.2.3, include: union, intersection, majority, and satisfied by $\geq N$. For this research question, we evaluate the effectiveness of our approach when using each of these heuristics. To simplify analysis, we do not vary the constraint templates and use the templates from the default configuration.

5.3.3 Results

RQ1: Effectiveness of SpecForge

The effectiveness of a specification mining approach affects the usefulness of the mined specification for program comprehension and for automated program analysis. To answer this research questions, I measure the precision, recall, and F-measure of SpecForge in inferring behavioral models of the 13 library classes de-

scribed in Section 5.3.1.

I execute SpecForge on an Intel(R) E5-2667 2.9 GHz processor server with 189 GB RAM running Linux 2.6; on average, SpecForge takes less than one second to infer a specification for each input class. Table 5.2 shows the precision, recall, and F-measure of SpecForge for the different target library classes. I note that precision ranges from 54.62% to 100%, and that recall ranges from 24.32% to 100%, while F-measure ranges from 39.13% to 100%. Noticeably, the precision is higher than the recall in most of the cases (11 out of 13 classes). Thus, most of the behaviors captured in the inferred models are correct but some correct behaviors are not captured successfully. Furthermore, there are 8 library classes for which my approach achieves a precision of 100%, and 1 library class for which its recall is 100%. My approach achieves the best F-measure for `LinkedList` (i.e., 90.59%) and it achieves the worst F-measure for `Signature` (i.e., 39.13%). Overall, SpecForge achieves an average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21%, respectively.

I have manually investigated the inaccuracies of models generated with SpecForge. I found that the main cause of the low F-measures is due to wrong temporal rules being selected and fused into the overall model. Sections 5.3.3 and 5.3.3 describe how I can further improve the F-measure.

RQ2: SpecForge vs. Baselines

Many specification mining approaches that analyze execution traces and output a finite state automaton have been proposed in the literature. SpecForge is built on top of 7 existing approaches proposed by Krka [46] (described in Section 2.2.1). To answer this research question, I compare the precision, recall, and F-measure of SpecForge with those of the 7 existing approaches [46] for each of the 13 library classes that I investigate in this work. These baselines are: traditional 1-tails, traditional 2-tails, CONTRACTOR++, SEKT 1-tails, SEKT 2-tails, optimistic

Table 5.3: Precision, Recall, and F-measure: Traditional 1-tail and Traditional 2-tail. “P” = Precision, “R” = Recall, and “F” = F-measure.

Target Library Class	Traditional 1-tails			Traditional 2-tails		
	P	R	F	P	R	F
ArrayList	100.00%	11.15%	20.06%	100.00%	10.45%	18.92%
HashMap	100.00%	22.68%	36.97%	100.00%	19.08%	32.05%
HashSet	100.00%	20.76%	34.38%	100.00%	13.50%	23.79%
Hashtable	100.00%	30.23%	46.43%	100.00%	21.89%	35.92%
LinkedList	100.00%	29.49%	45.55%	100.00%	26.49%	41.88%
StringTokenizer	68.18%	39.46%	49.99%	71.21%	21.77%	33.34%
NFST	100.00%	1.80%	3.54%	100.00%	0.90%	1.79%
SMTPProtocol	100.00%	17.50%	29.79%	100.00%	17.50%	29.79%
Signature	100.00%	8.11%	15.00%	100.00%	8.11%	15.00%
Socket	97.15%	10.18%	18.43%	98.69%	8.86%	16.26%
StackAr	34.04%	14.52%	20.36%	51.69%	14.52%	22.67%
ToHTMLStream	100.00%	20.00%	33.33%	100.00%	20.00%	33.33%
ZipOutputStream	100.00%	0.00%	0.00%	95.00%	0.00%	0.00%
Average	92.26%	17.38%	27.22%	93.58%	14.08%	23.44%

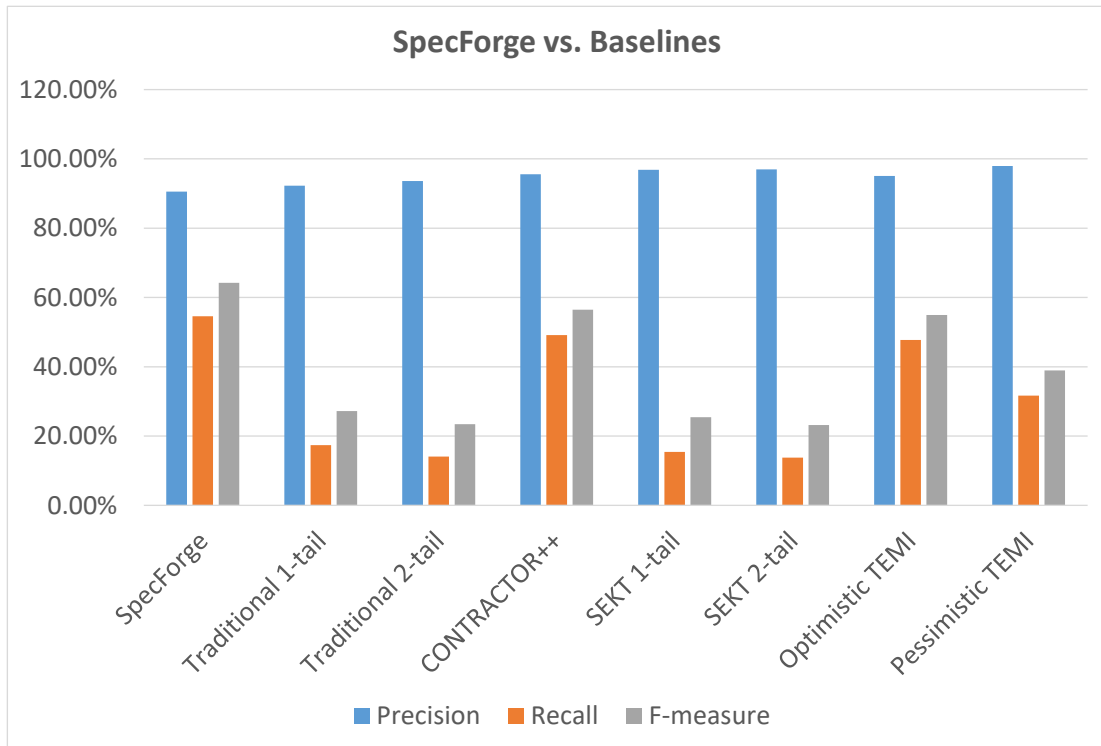


Figure 5.4: Average Precision, Recall, and F-measure: SpecForge vs. Baselines.

Table 5.4: Precision, Recall, and F-measure: CONTRACTOR++. “P” = Precision, “R” = Recall, and “F” = F-measure.

Target Library Class	CONTRACTOR++		
	P	R	F
ArrayList	100.00%	46.15%	63.15%
HashMap	100.00%	4.32%	8.28%
HashSet	100.00%	100.00%	100.00%
Hashtable	100.00%	1.55%	3.05%
LinkedList	100.00%	79.39%	88.51%
StringTokenizer	71.38%	16.33%	26.57%
NFST	87.27%	40.54%	55.36%
SMTProtocol	85.71%	50.00%	63.16%
Signature	100.00%	75.68%	86.15%
Socket	100.00%	0.22%	0.44%
StackAr	98.35%	100.00%	99.17%
ToHTMLStream	100.00%	100.00%	100.00%
ZipOutputStream	100.00%	25.00%	40.00%
Average	95.59%	49.17%	56.45%

TEMI, and pessimistic TEMI. Tables 5.3, 5.4, 5.5 and 5.6 show the precision, recall, and F-measure of the baselines.³ Table 5.3 shows the precision, recall, and F-measure of traditional 1-tails and traditional 2-tails, and Table 5.4 shows the precision, recall, and F-measure of CONTRACTOR++. Table 5.5 shows the precision, recall, and F-measure of SEKT 1-tails and SEKT 2-tails, and Table 5.5 shows the precision, recall, and F-measure of Optimistic TEMI and Pessimistic TEMI. Figure 5.4 shows a bar chart that highlights the comparison between SpecForge and all baselines. From the tables and the bar chart, I find that CONTRACTOR++ has the best average F-measure of 56.45%, and SEKT 2-tails has the least average F-measure of 23.18%.

Comparing Table 5.2 with Tables 5.3, 5.4, 5.5 and 5.6, my approach outperforms the average F-measures of all the baselines by 13.75% to 177.02%. The average precision of my approach is slightly lower than those of the baselines (by

³Krka et al. have also estimated the precision and recall of these approaches [46]. I use a more rigorous evaluation setting to generate sentences from inferred and ground truth models, i.e., 10-transition coverage (see Section 5.3.1). Therefore, the results shown in Tables 5.5 and 5.3 are different from the ones calculated by Krka et al. [46].

Table 5.5: Precision, Recall, and F-measure: SEKT 1-tail and SEKT 2-tail, Optimistic TEMI, and Pessimistic TEMI. “P” = Precision, “R” = Recall, and “F” = F-measure.

Target Library Class	SEKT 1-tails			SEKT 2-tails		
	P	R	F	P	R	F
ArrayList	100.00%	10.50%	19.00%	100.00%	10.28%	18.64%
HashMap	100.00%	21.75%	35.73%	100.00%	19.02%	31.96%
HashSet	100.00%	20.76%	34.38%	100.00%	13.50%	23.79%
Hashtable	100.00%	27.44%	43.06%	100.00%	20.79%	34.42%
LinkedList	100.00%	28.45%	44.30%	100.00%	25.96%	41.22%
StringTokenizer	63.64%	21.77%	32.44%	75.94%	20.41%	32.17%
NFST	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
SMTPProtocol	100.00%	17.50%	29.79%	100.00%	17.50%	29.79%
Signature	100.00%	8.11%	15.00%	100.00%	8.11%	15.00%
Socket	100.00%	10.11%	18.36%	100.00%	8.86%	16.28%
StackAr	95.55%	14.52%	25.21%	84.85%	14.52%	24.80%
ToHTMLStream	100.00%	20.00%	33.33%	100.00%	20.00%	33.33%
ZipOutputStream	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
Average	96.86%	15.45%	25.43%	96.98%	13.77%	23.18%

up to 8.11%), however its recall is substantially higher than those of the baselines (by up to 296.37%). Overall, the above statistics show that my approach is more effective than all of the baselines.

RQ3: Different Constraint Templates

In this paper, I consider six different constraint templates. Some constraints may capture important properties in a FSA that a miner successfully “generalizes” from a set of execution traces, while others may capture the idiosyncrasies of a FSA miner which “overfit” the execution traces. In this research question, I investigate if some constraint templates are prone to capturing incorrect constraints. To answer this question, I evaluate the effectiveness of SpecForge when only some of the constraint templates are considered. I then highlight the effect of adding and omitting some constraint templates.

Table 5.7 and Figure 5.5 compare the effectiveness of SpecForge when different constraint templates are used in the constraint enumeration step. Due to space

Table 5.6: Precision, Recall, and F-measure: Optimistic TEMI and Pessimistic TEMI. “P” = Precision, “R” = Recall, and “F” = F-measure.

Target Library Class	Optimistic TEMI			Pessimistic TEMI		
	P	R	F	P	R	F
ArrayList	100.00%	31.03%	47.36%	100.00%	18.92%	31.82%
HashMap	100.00%	4.32%	8.28%	100.00%	1.71%	3.36%
HashSet	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Hashtable	100.00%	0.16%	0.32%	100.00%	1.58%	3.11%
LinkedList	100.00%	79.39%	88.51%	100.00%	34.51%	51.31%
StringTokenizer	52.89%	14.29%	22.50%	78.99%	17.01%	27.99%
NFST	89.61%	40.54%	55.83%	94.00%	30.63%	46.21%
SMTPProtocol	94.81%	50.00%	65.47%	100.00%	5.00%	9.52%
Signature	100.00%	75.68%	86.15%	100.00%	75.68%	86.15%
Socket	100.00%	0.22%	0.44%	100.00%	18.00%	30.51%
StackAr	98.55%	100.00%	99.27%	100.00%	1.79%	3.53%
ToHTMLStream	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
ZipOutputStream	100.00%	25.00%	40.00%	100.00%	6.82%	12.77%
Average	95.07%	47.74%	54.93%	97.92%	31.67%	38.94%

constraints, I do not show all possible combinations. From the table, I note that there are several combinations of constraint templates that result in higher average precision, recall and F-measure compared to the default setting. Among the combinations shown in Table 5.7, AF + NF + AP and AF + NF + AP + AIF have the highest average precision, recall, and F-measure respectively, which are 83.35%, 71.82% and 72.82%.

On the other hand, ALL – AP and AIF + NIF + AIP have the least average F-measure of approximately 22%. The decrease in F-measure shows that the absence of the *always preceded by* constraints has a significant impact on the effectiveness of SpecForge. Overall, choosing a suitable combinations of constraint templates is important for improving effectiveness.

RQ4: Different Constraint Selection Heuristics

In this work, I propose a number of heuristics that I can adopt when selecting constraints that have been extracted from the input FSAs. These heuristics, presented in Section 5.2.3, include: union, intersection, majority, and satisfied by

Table 5.7: Average Precision, Recall, and F-measure: SpecForge with Different Constraint Templates.

Constraint Templates	Average		
	Precision	Recall	F-measure
ALL (default)	90.57%	54.58%	64.21%
ALL – AF	87.58%	60.52%	68.21%
ALL – NF	90.68%	54.98%	64.83%
ALL – AP	15.01%	54.58%	21.36%
ALL – AIF	90.73%	54.58%	64.33%
ALL – NIF	86.60%	62.62%	66.71%
ALL – AIP	89.85%	63.22%	70.75%
AF + NF + AP	83.35%	71.82%	72.82%
AF + NF + AP + AIP	86.57%	62.62%	66.70%
AF + NF + AP + NIF	89.85%	63.22%	70.75%
AF + NF + AP + AIF	83.35%	71.82%	72.82%
AIF + NIF + AIP	14.44%	60.92%	21.94%

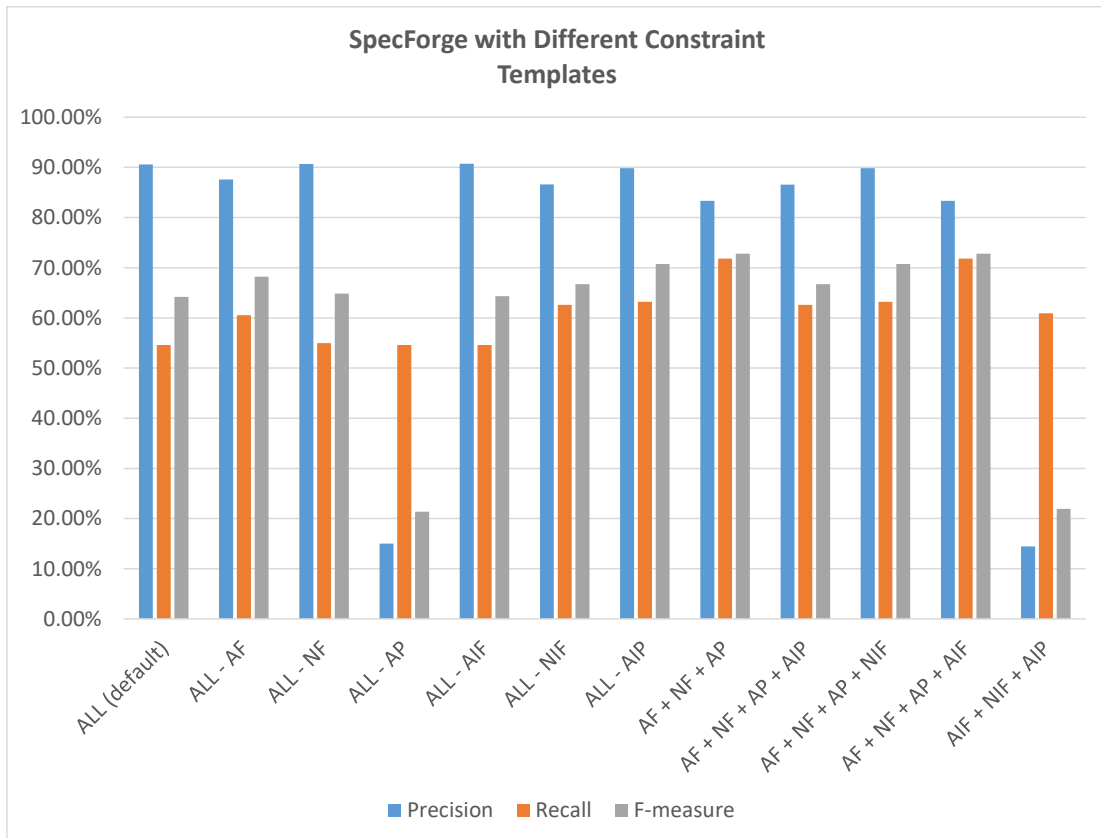


Figure 5.5: Average Precision, Recall, and F-measure: SpecForge with Different Constraint Templates.

Table 5.8: Average Precision, Recall, and F-measure: SpecForge with Different Constraint Selection Heuristics.

Selection Heuristics	Precision	Recall	F-measure
Union	56.19%	10.26%	15.40%
Satisfied By $\geq x = 2$	78.51%	12.01%	18.36%
Satisfied By $\geq x = 3$	83.62%	17.81%	25.36%
Majority	93.00%	20.24%	28.98%
Satisfied By $\geq x = 5$	89.80%	34.98%	45.34%
Satisfied By $\geq x = 6$	88.82%	48.56%	59.48%
Intersection (default)	90.57%	54.58%	64.21%

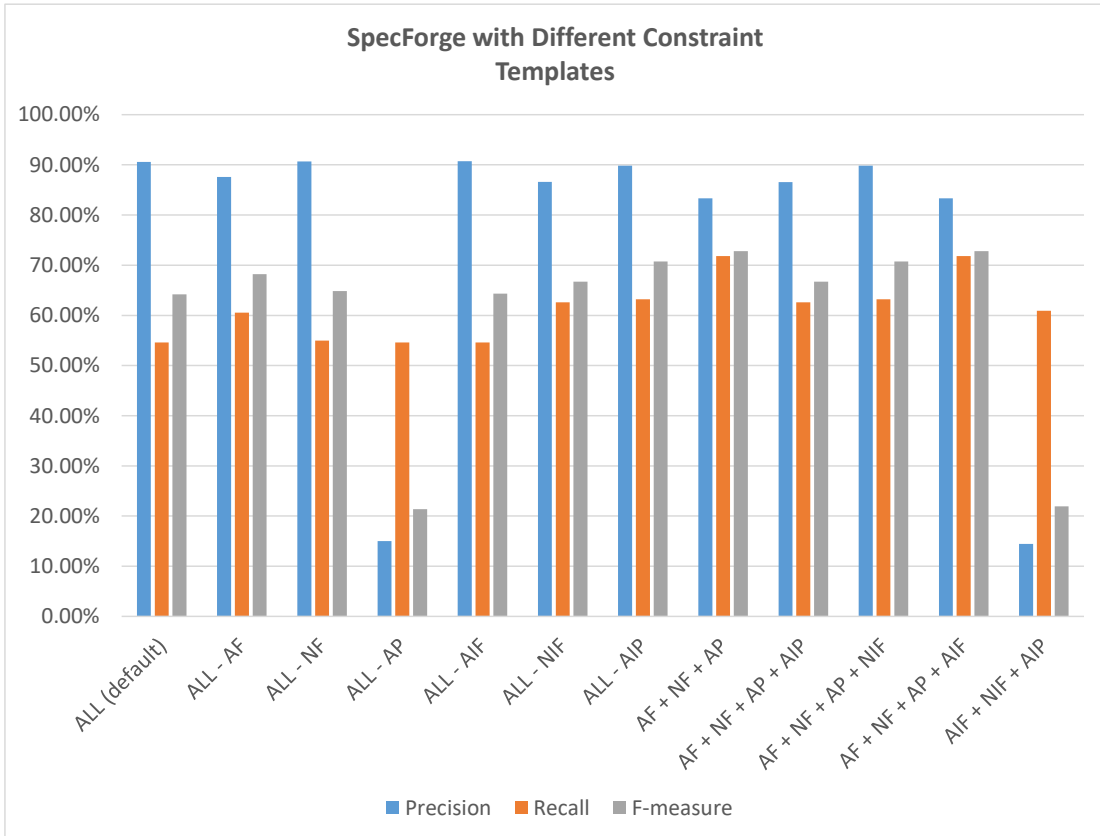


Figure 5.6: Average Precision, Recall, and F-measure: SpecForge with Different Constraint Selection Heuristics.

$\geq N$. For this research question, I evaluate the effectiveness of my approach when using each of these heuristics. To simplify analysis, I do not vary the constraint templates and use the templates from the default configuration.

Table 5.8 and Figure 5.6 compare SpecForge’s performance for different con-

straint selection heuristics. The table lists the selection heuristics in increasing order of strictness in selecting constraints. From the table, I notice that *intersection* is the most effective selection heuristic with an F-measure of 64.21%, while union is the least effective heuristic with an F-measure of 15.40%. The results also show that stricter selection heuristics tend to improve SpecForge’s F-measure. Note that with a stricter heuristic, SpecForge only selects a subset of constraints which enlarges the language accepted by the final inferred FSA. This potentially increases recall (if more correct sentences are accepted by the inferred FSA) – in the worst case recall will remain the same (if no additional correct sentences are accepted). Selecting fewer constraints may reduce precision (if many incorrect sentences are accepted by the inferred FSA). However, in the results, I note that precision, in general, increases with stricter heuristics – this shows that the additional sentences in the accepted languages of inferred FSAs include no or few incorrect sentences.

Discussion

Untapped Potentials. In this paper I evaluated SpecForge based on the 7 specification mining techniques analyzed by Krka et al. [46]. Currently, my meta-approach works better than existing baselines on average, however, it does not perform the best in all cases. In my future work, I plan to improve SpecForge’s performance in two concrete ways:

1. I plan to extend SpecForge with other specification mining techniques, such as Synoptic [15] and Perfume [87]. This can help SpecForge exclude more incorrect constraints, which is the main reason why currently SpecForge performs poorly on some target classes. As just one example, if I drop 4 incorrect constraints that are used to construct the FSA for `java.util.Hashtable` I can boost the F-measure of the inferred model from 61.22% to 72.30%.

2. I also plan to investigate alternative selection heuristics. Currently SpecForge applies the same constraint selection heuristic across all the constraint templates. However, it is possible to apply a different selection heuristic to each template. This will allow us to vary the strictness of the heuristics used for the constraint selection process. One promising direction is to use machine learning approaches to identify the best heuristics to apply for each template by learning over a large training dataset.

5.4 Conclusion

I have evaluated SpecForge by inferring specifications of 13 target library classes from the execution traces of their client applications. I demonstrated that the FSAs constructed with SpecForge are superior to those inferred by any one specification mining approach. My experiments show that SpecForge (with default configuration) can achieve an average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21% respectively. Although the average precision of SpecForge is slightly lower than the baselines (by up to 8.11%), its average recall is significantly better (by up to 296.37%). In terms of average F-measure, the harmonic mean of precision and recall, SpecForge improves over the best performing baseline by 13.75%. I have also tried to adjust the configuration of SpecForge, and the best configuration can achieve an average precision, recall, and F-measure of 83.35%, 71.82%, and 72.82% respectively. I believe that SpecForge generalizes and can easily include, and build on, other FSA specification mining approaches.

In the future, I plan to improve the effectiveness of SpecForge further by increasing the number of underlying FSA miners synergized together, increasing the number of constraint templates, and developing an approach that can infer good configurations of constraint templates and selection heuristics based on training data. I also plan to reduce the threats to external validity further by experimenting with additional target library classes and execution traces. Moreover, I plan to extend SpecForge to mine parametric specification following the work by Lee

et al. [54] and Lo et al. [25, 68].

Chapter 6

DEEP SPECIFICATION MINING

Formal specifications are essential but usually unavailable in software systems. Furthermore, writing these specifications is costly and requires skills from developers. Recently, many automated techniques have been proposed to mine specifications in various formats including finite-state automaton (FSA). However, more works in specification mining are needed to further improve the accuracy of the inferred specifications.

In this work, we propose Deep Specification Miner (DSM), a new approach that performs deep learning for mining FSA-based specifications. Our proposed approach uses test case generation to generate a rich set of execution traces for training a Recurrent Neural Network Based Language Model (RNNLM). From these execution traces, we construct a Prefix Tree Acceptor (PTA) and use the learned RNNLM to extract many features. These features are subsequently utilized by clustering algorithms to merge similar automata states in PTA for constructing more accurate resultant models. Finally, DSM performs a model selection heuristic to select the most accurate model as the final one. We execute DSM to mine specifications of 11 target library classes. Our empirical analysis shows that DSM achieves an average Precision, Recall, and F-measure of 82.76%, 72.3%, and 71.97%, respectively. Compared to the best baseline, our approach is more effective by 29.82% in terms of average F-measure.

6.1 Introduction

To mine more accurate FSA models, we propose a new specification mining algorithm that performs deep learning on execution traces. We name our approach as

DSM which stands for Dep Specification Miner. Recently, deep machine learning techniques are proposed to learn representations of data with multiple levels of abstraction [52]. Specifically, deep learning techniques have the capabilities to learn complex representations from large amount of data (e.g., execution traces) by utilizing deep neural networks (i.e., networks with many layers). In this work, we perform deep learning using Recurrent Neural Networks (RNN) [80] to learn a complex model from temporal behaviors of software systems encoded in execution traces. RNN is particularly good for learning models that depend on time or temporal information as RNN’s hidden layers are allowed to connect to each other. Our approach takes as input a target library class C and employs an automated test case generation tool to generate thousands of test cases. The goal of this test case generation process is to capture a rich set of valid sequences of invoked methods of C . Next, we perform deep learning on execution traces of generated test cases to train a Recurrent Neural Network Language Model (RNNLM) [80]. We construct a Prefix Tree Acceptor (PTA) from the execution traces and leverage the learned language model to extract a number of interesting features from PTA’s nodes. These features are then input to clustering algorithms for merging similar states (i.e., PTA’s nodes). The output of an application of a clustering algorithm is a simpler and more generalized FSA that reflects the training execution traces. Finally, our approach predicts the accuracy of constructed FSAs (generated by different clustering algorithms considering different settings) and outputs the one with highest predicted value of F-measure.

We evaluate our proposed approach for 11 target library classes which were used before to evaluate many prior work [46, 47]. For each of the input class, we first run Randoop to generate thousands of test cases. Then, we use execution traces of these test cases to infer FSAs. Our experiments show that DSM achieves an average precision, recall, and F-measure of 82.76%, 72.3%, and 71.97%, respectively. Compared to other existing specification mining algorithms, our ap-

proach outperforms all baselines that construct FSAs from execution traces (e.g., k-tails [16], SEKT, etc.) by at least 29.82% in terms of average F-measure. Some of the baselines first use Daikon to learn invariants that are then used to infer a better FSA. Our approach does not use Daikon invariants in the inference of FSA. Excluding baselines that use Daikon invariants, our approach can outperform the remaining best performing miner by 33.24% in terms of average F-measure.

The contributions of our work are highlighted below:

1. We propose DSM (Deep Specification Miner), a new specification mining algorithm that utilizes test case generation, deep learning, clustering algorithms, and model selection strategy to infer automaton based specifications. To the best of our knowledge, we are the first to use deep learning for mining specifications.
2. We employ deep neural network based language models to estimate a number of interesting features to be extracted from automaton states. These features are subsequently used to support clustering algorithms to merge similar states in order to infer simpler automata.
3. We propose a model selection strategy that predicts precision, recall, and F-measure of a FSA given a set of execution traces of generated test cases. Our approach selects and returns the FSA with highest predicted F-measures.
4. We evaluate the effectiveness of DSM on 11 different target library classes. Our results show that our approach outperforms the best baseline by 29.82% in terms of average F-measure.

The remainder of this paper is structured as follows. We describe details of our proposed approach in Section 6.2. Then, we present our empirical evaluation in Section 6.3. We discuss the future work and conclusion in Section 6.4.

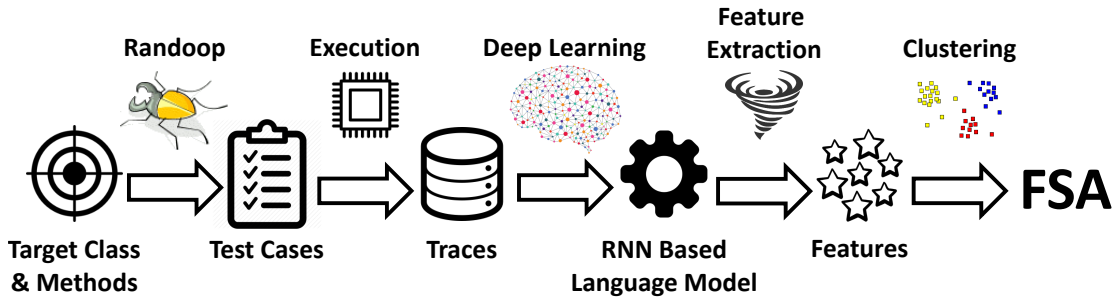


Figure 6.1: DSM’s overall framework.

6.2 Deep Specification Mining

6.2.1 Overall Framework

Figure 6.1 shows the overall framework of our proposed approach. In our framework, there are three major processes: test case generation and traces collection, Recurrent Neural Network Based Language Model (RNNLM) learning, and automata construction. Our approach takes as input a target class and signatures of methods. Then, DSM runs Randoop [98] to generate a substantial number of test cases for the input target class. Then, we record the execution of these test cases, and retain traces of invocations of methods of the input target class as the training dataset. Next, our approach performs deep learning on the collected traces to infer a RNNLM that is capable of predicting the next likely method to be executed given a sequence of previously called methods.

Subsequently, we employ a heuristic to select a subset of traces that best represents the whole training dataset. From these traces, we construct a Prefix Tree Acceptor (PTA); we refer to each PTA’s node as an automaton state. Utilizing the inferred RNNLM, we extract a number of features from automaton states, and input the feature values to a number of clustering algorithms (i.e., k-means [74] and hierarchical clustering [99]) considering different settings (e.g., different number of clusters). The output of a clustering algorithm is clusters of similar automaton states. We use these clusters to create a new FSA by merging states that belong

to the same cluster. Every application of a clustering algorithm with a particular setting results in a different FSA. We propose a model selection strategy to heuristically select the most accurate model by predicting values of Precision, Recall, and F-measure. Finally, we output the FSA with highest predicted F-measure.

6.2.2 Test Case Generation and Trace Collection

This process plays an important role to our approach as it decides the quality of RNNLM inferred by the deep learning process. Previous research works in specification mining [46, 49, 47] collect traces from the execution of a program given unit test cases or inputs manually created by researchers. In this work, we utilize deep learning for mining specification. Deep learning requires a substantially large and rich amount of data. The more training inputs, the more patterns the resultant RNNLM can capture. In general, it is difficult to follow previous works to collect a rich enough set of execution traces for an arbitrary target library class. Firstly, it is challenging to look for all projects that use the target library class, especially for classes from new or unreleased libraries. Secondly, existing unit test cases or manually created inputs may not cover many of the possible execution scenarios of methods in a target class.

We address the above issues by following Dallmeier et al. [23, 22] to generate as many test cases as possible for mining specifications, and collect the execution traces of these test cases for subsequent steps. Recently, many test case generation tools have been proposed such as Randoop¹ [98], EvoSuite² [33], etc. Among the state-of-the-art test case generation tools, we choose Randoop because it is widely used and lightweight. Furthermore, Randoop is well maintained and frequently updated with new versions. As future work, we plan to integrate many other test case generation methods into our approach.

Randoop generates a large number of test cases, which is proportional to the

¹<https://randoop.github.io/randoop/>

²<http://www.evosuite.org/>

time limit of its execution. In order to improve the coverage of possible sequences of methods under test, we provide class-specific literals aside from default ones to Randoop. For example, for `java.net.Socket`, we create string and integer literals which are addresses of hosts (e.g., “localhost”, “127.0.0.1”, etc.) and listening ports (e.g., 8888, etc.). Furthermore, we create driver classes that contain static methods that invoke constructors of the target class to initialize new objects. That helps speed up Randoop to create new objects without spending time to search for appropriate input values for constructors.

6.2.3 Learning RNNLM for Specification Mining

In this section, we first briefly discuss statistical language model in general. Next, we present an overview of Recurrent Neural Network Based Language Models (RNNLM), which is a specific type of language model used in this work. DSM employs RNNLM to extract several important features from states of an automaton, and use these features to merge similar states.

Statistical Language Model: A statistical language model is an oracle that can foresee how likely a sentence $\mathbf{s} = w_1, w_2, \dots, w_n$ to occur in a language. In a nutshell, a statistical language model considers a sequence \mathbf{s} to be a list of words w_1, w_2, \dots, w_n and assigns probability to \mathbf{s} by computing joint probability of words: $P(w_1, \dots, w_n) = \prod_{i=1}^{n-1} P(w_i | w_1, \dots, w_{i-1})$. As it is challenging to compute conditional probability $P(w_i | w_1, \dots, w_{i-1})$, different language model has its own assumption to approximate the calculation. N-grams model, a popular family of language models, approximates in a way that a word w_k conditionally depends only on its previous N words (i.e., $w_{k-N+1}, \dots, w_{k-1}$). For example, unigram model simply estimates $P(w_i | w_1, \dots, w_{i-1})$ as $P(w_i)$, bigram model approximates $P(w_i | w_1, \dots, w_{i-1})$ as $P(w_i | w_{i-1})$, etc. In this work, we utilize the ability of language models to compute $P(w_i | w_1, \dots, w_{i-1})$ for estimating features of automaton states. We consider every method invocation as a word and an execution trace of

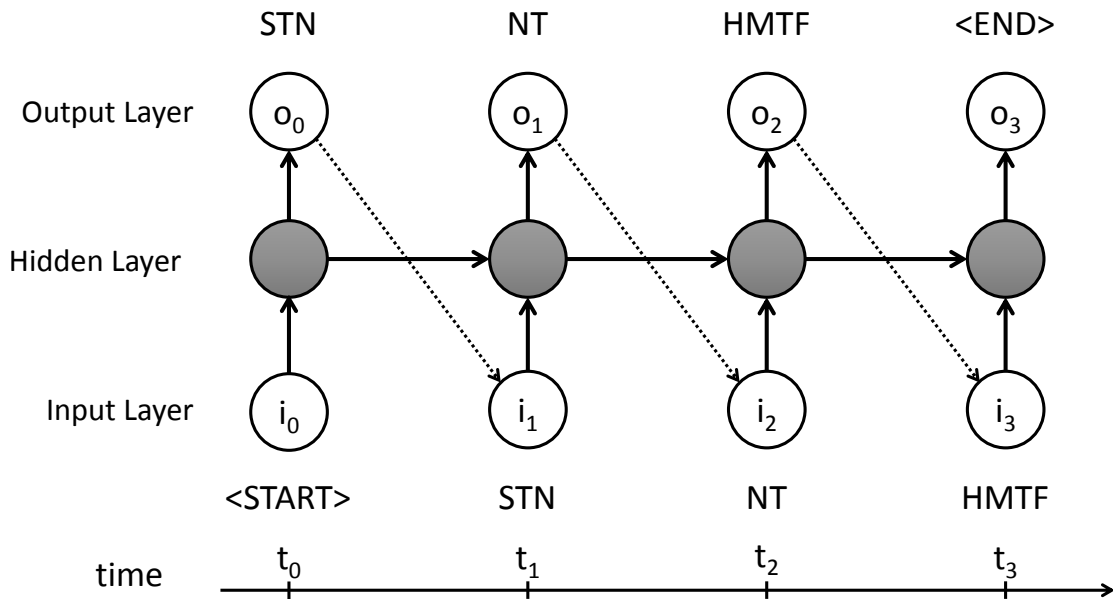


Figure 6.2: An unrolled Recurrent Neural Network from time t_0 to t_3 for predicting the next likely method given a sequence of invoked methods for `java.util.StringTokenizer`. “STN” : `StringTokenizer()`, “NT” : `nextToken()`, and “HMTF” : `hasMoreTokens()==false`.

an object as a sentence (i.e., sequence of method invocations). Given a sequence of previously invoked methods, we use a language model to output the probability of a method to be invoked next.

Recurrent Neural Network Based Language Model: N-grams models have been widely used to solve many software engineering tasks. For decades, there have been many research studies trying to outperform N-grams. Recently, a family of language models that make use of neural networks is shown to be more effective than N-grams [79]. These models are referred to as neural network based language models (NNLM). If a NNLM has many hidden layers, we refer to the model as a *deep neural network language model* or *deep language model* for short. Furthermore, the process of training weights in the underlying network of such deep language models is called *deep learning*. Recently, Recurrent Neural Network Based Language Model (RNNLM) [80] is well-known with its ability to use internal memories to handle sequences of words with arbitrary lengths. The un-

derlying network architecture of a RNNLM is a Recurrent Neural Network (RNN) that stores information of input word sequences in its hidden layers. Figure 6.2 demonstrates how a RNN operates given the sequence `<START>`, `STN`, `NT`, `HMTF`, `<END>`. In the figure, a RNN is unrolled to become four connected networks, each of which is processing one input method at a time step. Initially, all states in the hidden layer are assign to all zeros. At time t_k , a method m_k is represented as an one-hot vector i_k by the input layer. Next, the hidden layer updates its states by using the vector i_k and the states previously computed at time t_{k-1} . Then, the output layer estimates a probability vector o_k across all methods for them to appear in the next time step t_{k+1} . This process is repeated at subsequent time steps until the last method in the sequence is handled.

Construction of Training Method Sequences: Our set of collected execution traces is a series of method sequences. Each of these sequences starts and ends with two special symbol: `<START>` and `<END>`, respectively. These symbols are used for separating two different sequences. We gather all sequences together to create data for training Recurrent Neural Network. Furthermore, we limit the maximum frequency of a method sequence `MAX_SEQ_FREQ` to 10 to prevent imbalanced data issue where a sequence appears much more frequently than the other ones.

Model Training: We perform deep learning on the training data to learn a Recurrent Neural Network Based Language Model (RNNLM) for every target library class. By default, we use Long Short-Term Memory (LSTM) network [37], one of the state-of-the-art RNNs, as the underlying architecture of the RNNLM. Compared to the standard RNN architecture, LSTM is better in learning long-term dependencies. In fact, LSTM network is more suitable for learning from execution traces data as sequences of methods in the traces are usually long.

6.2.4 Automata Construction

In this processing step, our approach takes as input the set of training execution traces and the inferred RNNLM (see Section 6.2.3). The output of this step is a FSA that best captures the specification of the corresponding target class. The construction of FSA undergoes several substeps: trace sampling, feature extraction, clustering, and model selection.

At first, we use a heuristic to select a subset of method sequences that represents all training execution traces. The feature extraction and clustering steps use these selected traces, instead of all traces, to reduce computation cost. We construct a Prefix Tree Acceptor (PTA) from the selected traces and extract features for every PTA nodes using the inferred RNNLM. We refer to each PTA node as an automata state. Figure 6.3 shows an example of a PTA constructed from three different sequences of methods of `java.util.ZipOutputStream`. We want to find similar automata states and group them into the same cluster. In the clustering substep, we run a number of clustering algorithms on PTA nodes with various settings to create many different FSAs. Finally, in the model selection substep, we follow a heuristic to predict the F-measure (see Section 6.3.2) of constructed FSAs and output the one with highest predicted F-measure. The full set of traces is used in this model selection step. In the following paragraphs, we describe details of each substep in this processing step:

Trace Sampling: Our training data contains a large number of sequences. Thus, it is expensive to use all of them for constructing FSAs. Therefore, the goal of trace sampling is to create a smaller subset that is likely to represent the whole set of all traces reasonably well. We propose a heuristic to find a subset of traces that covers all co-occurrence pairs of methods in all training traces. (m_1, m_2) is a co-occurrence pair if m_1 and m_2 appear together in at least one trace.

Algorithm 3 shows our heuristic to select execution traces from the whole set of execution traces S . At first, we determine all pairs (a, b) where a and b

Algorithm 3 Selecting Subset of Method Sequences

Require: $S = \{S_i \mid 1 \leq i \leq N\}$: Collection of method sequences where N is the number of sequences

Ensure: X : Selected sequences of methods

```
1: Sort  $S$  in ascending order of sequence length
2:  $D \leftarrow$  initialization of dictionary type
3:  $P \leftarrow$  empty set
4: for  $S_i \in S$  do
5:   for  $(a, b)$  where  $a \in S_i \wedge b \in S_i \wedge a < b$  do
6:      $D[(a, b)] += [S_i]$ 
7:     Include  $(a, b)$  to  $P$ 
8:   end for
9: end for
10:  $X \leftarrow$  empty set
11: while  $\exists(a, b) \in P$  do
12:   Select  $(a, b) \in P$  that  $D[(a, b)]$  has the least number of elements
13:   for  $S_i \in D[(a, b)] \wedge S_i \notin X$  do
14:     Include  $S_i$  to  $X$ 
15:     Remove all pairs  $(a, b)$  from  $P$  where  $a \in S_i \wedge b \in S_i \wedge a < b$ 
16:   break
17: end for
18: end while
19: return  $X$ 
```

are methods that occur together in at least one trace in S and store them in P (lines 4 to 9). Next, we create a set O that contains selected traces – initially O is an empty set (line 10). Then, we iteratively choose a pair (a, b) which does not appear in any trace in O and occur in the least number of input traces in S (line 12). Given a selected pair (a, b) , we look for the shortest trace $S_i \notin O$ where $a, b \in S_i$ (line 13). Once the trace is found, we include S_i to O . We mark the pair (a, b) as processed by removing it from P (line 15). We keep searching for sequences until O covers all co-occurrence pairs (a, b) in the input execution traces.

Feature Extraction: From method sequences of the sampled execution traces, we construct a Prefix Tree Acceptor (PTA). A PTA is a tree-like deterministic finite automaton (DFA) created by putting all the prefixes of sequences as states, and a PTA only accepts the sequences that it is built from. The final states

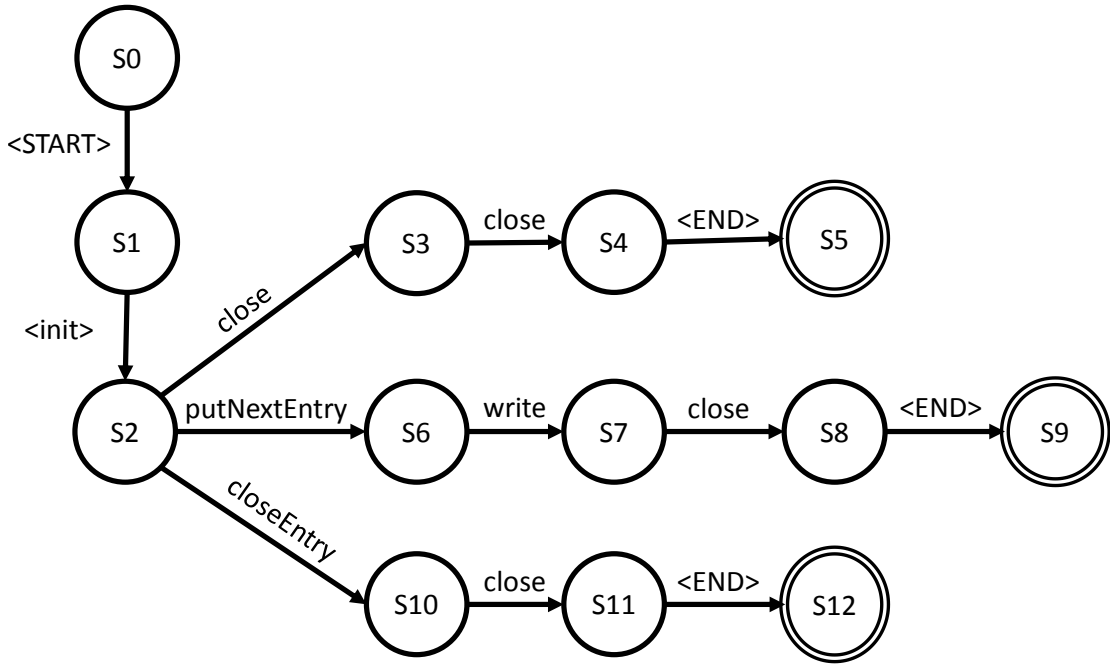


Figure 6.3: An example Prefix Tree Acceptor (PTA).

of our constructed PTAs are the ones have incoming edges with `<END>` labels (see Section 6.2.3). Figure 6.3 shows an example of a Prefix Tree Acceptor (PTA). Table 6.1 shows information of the extracted features. For each state S of a PTA, we are particularly interested in two types of features:

1. Type I: This type of features captures information of previously invoked methods before the state S is reached. The values of type I features for state S is the occurrences of methods on the path between the starting state (i.e., the root of the PTA) and S . For example, according to Figure 6.3, the values of Type 1 features corresponding to node S_3 are: $F_{\langle \text{START} \rangle} = F_{\langle \text{init} \rangle} = F_{\text{close}} = 1$ and $F_{\text{write}} = F_{\text{putNextEntry}} = F_{\text{closeEntry}} = F_{\langle \text{END} \rangle} = 0$.
2. Type II: This type of features captures the likely methods to be called after a state is reached. Values of these features are computed by the inferred RNNLM in the deep learning step (see Section 6.2.3). For example, at node S_3 in Figure 6.3, `close` and `<END>` have higher probabilities than the other

Table 6.1: Extracted Features for An Automata State

Feature ID	Value
Type I: Previously Invoked Methods	
F_m	1 if m is invoked before the automata state. Otherwise, 0.
Type II: Next Methods to Invoke	
P_m	Probability computed by the learned Recurrent Neural Network based Language Model for method m to be called after a particular automata state is reached ($0 \leq P_m \leq 1$).

methods to be called afterward. Examples of type II features and their values for node S_3 output by a RNNLM are as follows: $P_{\text{close}} = P_{\langle \text{END} \rangle} = 0.4$ and $P_{\langle \text{START} \rangle} = P_{\langle \text{init} \rangle} = P_{\text{write}} = P_{\text{putNextEntry}} = P_{\text{closeEntry}} = 0.04$.

Our intuition of extracting different types of features is to provide sufficient information for clustering algorithms in the subsequent substep to better merge PTA nodes.

Clustering: We run k-means [74] and hierarchical clustering [99] algorithms on the PTA’s states with their extracted features. Our goal is to create a simpler and more generalized automaton that captures specifications of a target library class. Since both k-means and hierarchical clustering require the predefined input C for number of clusters, we try with many values of C from 2 to `MAX_CLUSTER` (refer to Section 6.3.2) to search for the best FSA. Overall, the execution of clustering algorithms results in $2 \times (\text{MAX_CLUSTER} - 1)$ FSAs.

Model Selection: We propose a heuristic to select the best FSA among the ones output by the clustering algorithms. Algorithm 4 describes our strategy to predict precision of an automaton M given the set of all traces $Data$ (see Section 6.2.2).

We predict Precision by first constructing a set P_{Data} containing all pairs (m_1, m_2) , where m_1 and m_2 appear consecutively (i.e., m_1 is called right before m_2) in an execution trace in $Data$. Then, we construct another set P_M containing

Algorithm 4 Predicting Precision of a finite-state automaton given a set of method sequence.

Require:

M : an finite-state automaton

$Data$: a set of training method sequences

Ensure: Predicted Precision of M

```

1:  $P_{Data} \leftarrow$  empty set
2: for  $seq \in Data$  do
3:   for  $0 \leq i < \text{length}(seq) - 1$  do
4:     Include  $(seq[i], seq[i + 1])$  to  $P_{Data}$ 
5:   end for
6: end for
7:  $P_M \leftarrow$  empty set
8:  $E_M \leftarrow$  the set of transitions in  $M$ 
9: for  $s_1 \xrightarrow{m_1} s_2 \in E_M \wedge s_2 \xrightarrow{m_2} s_3 \in E_M$  do Include  $(m_1, m_2)$  to  $P_M$ 
10: end for
11:  $precision \leftarrow \frac{|P_{Data}|}{|P_{Data} \cup P_M|}$ 
12: return  $precision$ 

```

all pairs (m_1, m_2) that appear consecutively in a trace generated by the automaton M . In Algorithm 4, lines 1 to 6 compute all pairs (m_1, m_2) occurring in $Data$, and lines 7 to 11 collect those pairs occurring in traces generated by the input automaton M . To find all pairs occurring in traces generated by M , we look for two transitions $s_1 \xrightarrow{m_1} s'_1$ and $s_2 \xrightarrow{m_2} s'_2$ of M , where $s'_1 = s_2$. We take labels of the two transitions (i.e., m_1 and m_2) and add a pair of methods (m_1, m_2) to the set P_M . Line 12 computes the predicted value of Precision, which is the ratio between of number of all pairs in P_M and all pairs in $P_{Data} \cup P_M$. We input all FSAs created by clustering algorithms and all execution traces to Algorithm 4 for estimating the FSAs' Precision.

Next, we approximate the values of Recall by computing the percentage of all execution traces accepted by a given automaton M . Once all precision and recall of FSA models are predicted, we compute the expected value of F-measure (i.e., harmonic mean of precision and recall) for each of the automata. Finally, our approach returns the FSA with highest expected F-measure.

Table 6.2: Target Library Classes. “# Methods” represents the number of class methods that are analyzed, “# Generated Test Cases” is the number of test cases generated by Randoop, “# Recorded Method Calls” is the number of recorded method calls in the execution traces, “NFST” stands for `NumberFormatStringTokenizer`.

Target Library Class	# Methods	# Generated Test Cases	# Recorded Method Calls
<code>ArrayList</code>	18	42,865	22,996
<code>HashMap</code>	11	53,396	67,942
<code>Hashtable</code>	8	79,403	89,811
<code>HashSet</code>	8	23,181	257,428
<code>LinkedList</code>	7	13,731	4,847
NFST	5	15,8998	95,149
<code>Signature</code>	5	79,096	205,386
<code>Socket</code>	21	80,035	130,876
<code>StringTokenizer</code>	5	148,649	336,924
<code>StackAr</code>	7	549,648	13,2826
<code>ZipOutputStream</code>	5	162,971	43,626

6.3 Evaluation

6.3.1 Dataset

Target Library Classes: In our experiments, we select 11 target library classes as the benchmark to evaluate the effectiveness of our proposed approach. These library classes were investigated by previous research works in specification mining [46, 47]. Table 6.2 shows further details of the selected library classes including information of collected execution traces. Among these library classes, 9 out of 11 are from Java Development Kit (JDK); the other two library classes are `DataStructure.StackAr` (from Daikon project) and `NumberFormatStringTokenizer` (from Apache Xalan). For every library class, we consider methods that were analyzed by Krka et al. [46].

Ground Truth Models: We utilize ground truth models created by Krka et al. [46]. Among the investigated library classes, we refine ground truth models of five Java’s Collection based library classes (i.e., `ArrayList`, `LinkedList`, `HashMap`, `HashSet`,

and `Hashtable`) to capture “empty” and “non-empty” states of `Collection` objects. We also revise ground truth models of `NumberFormatStringTokenizer` and `Socket` by including missing transitions of the original models.

6.3.2 Experimental Settings

Evaluation Metrics

We follow Lo and Khoo’s method [63] to measure precision and recall for assessing the effectiveness of our proposed approach. Lo and Khoo’s method has been widely adopted by many prior specification mining works [46, 49]. By definition, precision of an inferred FSA is the percentage of sentences (i.e., execution traces) accepted by its corresponding ground truth model among the ones that are generated by that FSA. Similarly, recall of an inferred FSA is the percentage of sentences accepted by itself among the ones that are generated by the corresponding ground truth model. If precision of a FSA is low, it means that the automaton generates many invalid sentences. Therefore, the higher the precision, the more likely sentences generated by the FSA are correct. On the other hand, if recall is low, the FSA overfits the training data and unable to accept many other valid sentences. Thus, the higher the recall, the more general the inferred FSA. We use F-measure, which is the harmonic mean of precision and recall, as a summary metric to evaluate specification mining algorithms. F-measure is defined as follows:

$$F\text{-Measure} = 2 \times \frac{\textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}} \quad (6.1)$$

Experimental Configurations & Environments

Randoop Configuration. In test case generation step, for each target class, we repeatedly execute Randoop (version 3.1.2) with a time limit of 5 minute with 20 different initial seeds. We set the time limit to 5 minutes to make sure

subsequent collected execution traces are not too long as well as not too short. We repeat execution of Randoop 20 times to maximize the coverage of possible sequences of program methods in Randoop generated test cases. Furthermore, we turn off Randoop’s option of generating error-revealing test cases (i.e., `--no-error-revealing-tests` is set to `true`) as executions of these test cases are usually interrupted by exceptions or errors, which results in incomplete method sequences for subsequent deep learning process.

RNNLM. We use a publicly available implementation of RNNLM based on TensorFlow (r.11.0).³ We execute this implementation on a NVIDIA DGX-1 Deep Learning system. We use the default configuration included as part of the implementation.

Clustering Configuration. In clustering step, we run k-means and hierarchical clustering with the setting of number of clusters from 2 to `MAX_CLUSTER = 20` for every target class. We use `sklearn.cluster.KMeans` and `sklearn.cluster.AgglomerativeClustering` of scikit-learn (version 0.18) with default settings.

Baselines

In the experiments, we compare the effectiveness of DSM with many previous specification mining works. Krka et al. propose a number of algorithms that analyze execution traces to infer FSAs [46]. These algorithms are k-tails, CONTRACTOR++, SEKT, and TEMI. CONTRACTOR++, TEMI, and SEKT infer models leveraging invariants learned using Daikon. On the other hand, k-tails construct models only from ordering of methods in execution traces. Despite the fact that DSM is not processing likely invariants, we include CONTRACTOR++, SEKT, and TEMI as baselines to compare the applicability of deep learning and

³<https://github.com/hunkim/word-rnn-tensorflow>

likely invariant inference in specification mining. For k-tails and SEKT, we choose $k \in \{1, 2\}$ following Krka et al. [46] and Le et al. [47]’s configurations. In total, we have six different baselines: Traditional 1-tails, Traditional 2-tails, CONTRACTOR++, SEKT 1-tails, SEKT 2-tails, and TEMI

We use the implementation of provided by Krka et al.⁴ [46]. We utilize Daikon [30] to collect execution traces from Randoop generated test cases and inferred likely invariants from all of the traces. Originally, Krka et al.’s implementation uses a 32-bit version of Yices 1.0 Java Lite API⁵, which only works with 32-bit Java Virtual Machine and limited to use up to 4 GB heap memory. Since the amount of execution traces is large, we follow two experimental schemes to run Krka et al.’s code:

1. **Scheme I:** Krka et al.’s implementation is updated to work with the 64-bit libraries of Yices 1 SMT solver⁶. Then, we input execution traces of all generated test case as well as Daikon invariants inferred by these traces to all baselines. For each application of Krka et al.’s code, we set the maximum allocated memory to 7 GB and time limit to 12 hours.
2. **Scheme II:** We use the original Krka et al.’s implementation and a set of execution traces corresponding to test cases generated by Randoop with *one* specific seed.

6.3.3 Research Questions

RQ1: How effective is DSM? In this research question, we compute precision, recall, and F-measure of FSAs inferred by our approach for the 11 target library classes.

RQ2: How does DSM compare to existing specification mining algorithms? In this research question, we compare DSM with a number of existing

⁴<http://softarch.usc.edu/wiki/doku.php?id=inference:start>

⁵http://atlantis.seidenberg.pace.edu/wiki/lep/Yices_Java_API_Lite

⁶<http://yices.csl.sri.com/old/download-yices1.shtml>

Table 6.3: Precision, Recall, and F-measure: DSM

Target Library Classes	Precision	Recall	F-measure
ArrayList	54.39%	13.95%	22.21%
HashMap	100.00%	76.53%	86.71%
HashSet	100.00%	62.39%	76.84%
Hashtable	100.00%	66.55%	79.92%
LinkedList	100.00%	18.33%	30.98%
NFST	75.18%	80.01%	77.52%
Signature	100.00%	100.00%	100.00%
Socket	41.69%	77.58%	54.24%
StackAr	59.21%	100.00%	74.38%
StringTokenizer	100.00%	100.00%	100.00%
ZipOutputStream	79.89%	100.00%	88.82%
Average	82.76%	72.3%	71.97%

specification mining algorithms proposed by Krka et al. [46] in various experimental schemes.

RQ3: Which Recurrent Neural Network (RNN) is best for DSM? By default, DSM trains RNNLMs using Long-Short Term Memory (LSTM) networks from execution traces. In this research question, we first adapt DSM to use the standard RNN and Gated Recurrent Units (GRU)[18] networks for constructing language models with the same learning configuration as LSTM (see Section 6.2.3). Then, we analyze the effectiveness of DSM for each neural network architecture (i.e., Standard RNN, LSTM, and GRU).

6.3.4 Findings

RQ1: DSM’s Effectiveness. Table 6.3 shows the precision, recall, and F-measure of DSM for the eleven target library classes (Section 6.3.1). According to the table, our approach achieves the average precision, recall, and F-measure of 82.76%, 72.30%, and 71.97%, respectively. Noticeably, for `StringTokenizer` and `Signature`, DSM infers models that are exactly matched to the ground truth models (i.e. F-measure of 100%). There are other 6 out of 11 library classes where

Table 6.4: Precision, Recall, and F-measure: Traditional 1-tail and Traditional 2-tail. “P” = Precision, “R” = Recall, and “F” = F-measure.

Target Library Class	Traditional 1-tails			Traditional 2-tails		
	P	R	F	P	R	F
ArrayList	70.33%	2.92%	5.61%	93.14%	1.57%	3.08%
HashMap	43.12%	32.95%	37.35%	57.05%	13.58%	21.93%
HashSet	57.01%	14.38%	22.96%	66.52%	8.67%	15.35%
Hashtable	76.68%	80.24%	78.42%	94.28%	60.05%	73.37%
LinkedList	100.00%	15.14%	26.30%	100.00%	4.20%	8.07%
NFST	86.00%	57.22%	68.72%	90.82%	35.49%	51.04%
SMTProtocol	85.56%	31.37%	45.91%	100.00%	27.55%	43.20%
Signature	100.00%	100.00%	100.00%	100.00%	91.22%	95.41%
Socket	48.51%	30.30%	37.30%	82.03%	12.69%	21.98%
StackAr	89.93%	27.88%	42.57%	89.93%	27.88%	42.57%
StringTokenizer	100.00%	100.00%	100.00%	100.00%	81.31%	89.69%
ZipOutputStream	70.23%	100.00%	82.51%	74.05%	82.57%	78.08%
Average	77.28%	49.37%	53.97%	87.32%	37.23%	45.31%

our approach achieves F-measure of 70% or greater. DSM has the least F-measure on `ArrayList` (i.e., 22.21%) and `LinkedList` (i.e., 30.98%). For both of the two least effective cases, we note that precision is higher than recall. This is due to the poor coverage of possible sequences of method calls in generated test cases which makes RNNLM overfitted. As future work, we plan to design a better test case generation technique for specification mining.

RQ2: DSM vs. Previous Works. Table 6.4, 6.5, and 6.6 show the precision, recall, and F-measure of 6 baselines proposed by Krka et al. [46]. Optimistic TEMI⁷ is the best performing baseline in terms of average F-measure; Traditional 1-tails is the best baseline that constructs models from execution traces with the average F-measure of 53.97%. According to the bar chart shown in Figure 6.4, we note that the average recall, and F-measure of DSM is higher than those of all the six baselines. That indicates our inferred models are more generalized and less overfitted. In terms of average F-measure, our approach outperforms the best baseline (i.e., Optimistic TEMI) by 29.82%. DSM is also more effective

⁷Krka’s implementation was not able to return any Optimistic TEMI models for all `StackAr`’s input execution traces.

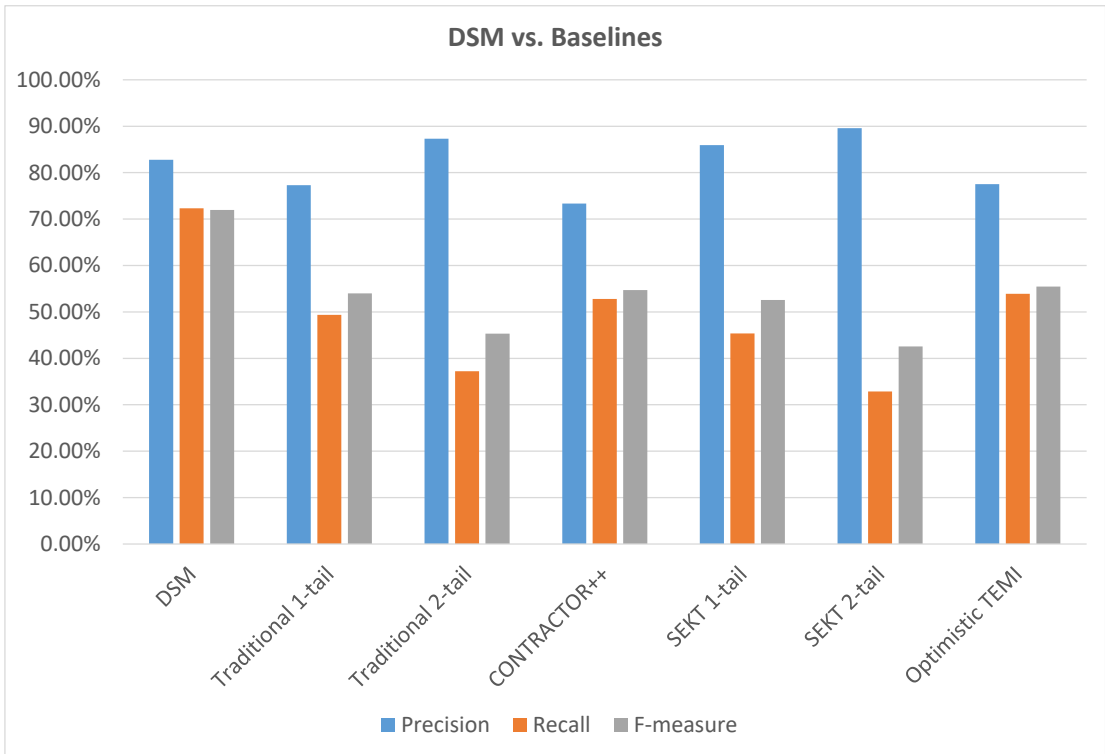


Figure 6.4: Average Precision, Recall, and F-measure: DSM vs. Baselines.

than Traditional 1-tails (i.e., the best baseline that infers FSAs from traces) by 33.24%. Noticeably, compared to the baselines that construct automata from execution traces (i.e., Traditional 1-tails, Traditional 2-tails, SEKT 1-tails, and SEKT 2-tails), DSM’s F-measures are higher for all target library classes. For the other baselines that use likely invariants, our approach is more effective in terms of F-measures for 7 out of 11 target library classes except `ArrayList`, `LinkedList`, `Hashtable`, and `Socket`.

RQ3: Best RNN Architecture. Table 6.7 and 6.8 shows the effectiveness of DSM configured with standard RNN architecture DSM^{RNN} and DSM^{GRU} , respectively. According to the tables, DSM^{GRU} outperforms DSM^{RNN} in terms of F-measure by 6.88%. GRU performs equally well as LSTM (our default setting).

Table 6.5: Precision, Recall, and F-measure: SEKT 1-tail and SEKT 2-tail. “P” = Precision, “R” = Recall, and “F” = F-measure, “N/A” = the result is not available.

Target Library Class	SEKT 1-tails			SEKT 2-tails		
	P	R	F	P	R	F
ArrayList	100.00%	2.22%	4.34%	100.00%	1.57%	3.08%
HashMap	43.12%	32.95%	37.35%	57.05%	13.58%	21.93%
HashSet	57.01%	14.38%	22.96%	66.52%	8.67%	15.35%
Hashtable	100.00%	69.05%	81.69%	100.00%	48.66%	65.47%
LinkedList	100.00%	10.16%	18.45%	100.00%	3.41%	6.59%
NFST	79.60%	57.22%	66.58%	92.89%	33.75%	49.51%
SMTPProtocol	100.00%	31.37%	47.76%	100.00%	27.55%	43.20%
Signature	100.00%	91.22%	95.41%	100.00%	81.46%	89.78%
Socket	85.17%	22.28%	35.32%	94.29%	11.73%	20.87%
StackAr	89.93%	27.88%	42.57%	89.93%	27.88%	42.57%
StringTokenizer	100.00%	85.54%	92.21%	100.00%	73.57%	84.77%
ZipOutputStream	76.16%	100.00%	86.47%	74.57%	62.23%	67.84%
Average	85.92%	45.36%	52.59%	89.60%	32.84%	42.58%

6.3.5 Discussions

According to the empirical evaluation, DSM outperforms all baseline mining algorithms that only analyze method ordering in execution traces to construct FSAs (i.e., k-tails). On the other hand, DSM is not better than CONTRACTOR++ and Optimistic TEMI for ArrayList, LinkedList, Hashtable, and Socket. Both of CONTRACTOR++ and Optimistic TEMI mainly rely on likely invariants rather than execution traces to construct FSAs. In fact, the two baselines benefit from accurate program invariants inferred from the execution of substantial amount of generated test cases (see Table 6.2).

In a nutshell, there is a trade-off between using likely invariants and raw method orderings in execution traces to construct FSAs. Program invariants are more helpful in inferring specifications [46], but they are more expensive to process and infer. This is because we need to record additional information in the traces such as values of all visible variables in entry and exit points of every invoked method. The more values of variables captured in the execution traces, the

Table 6.6: Precision, Recall, and F-measure: CONTRACTOR++ and Optimistic TEMI. “P” = Precision, “R” = Recall, and “F” = F-measure, “N/A” = the result is not available.

Target Library Class	CONTRACTOR++			Optimistic TEMI		
	P	R	F	P	R	F
ArrayList	86.30%	22.77%	36.03%	92.72%	22.19%	35.82%
HashMap	52.60%	100.00%	68.94%	52.60%	100.00%	68.94%
HashSet	57.11%	48.11%	52.22%	65.91%	48.11%	55.62%
Hashtable	100.00%	86.53%	92.78%	100.00%	86.53%	92.78%
LinkedList	100.00%	75.47%	86.02%	100.00%	75.47%	86.02%
NFST	29.00%	31.94%	30.40%	35.00%	31.94%	33.40%
SMTPProtocol	89.25%	33.62%	48.84%	99.66%	32.09%	48.54%
Signature	100.00%	50.24%	66.88%	100.00%	50.24%	66.88%
Socket	67.91%	46.43%	55.15%	69.37%	46.43%	55.62%
StackAr	52.03%	26.26%	34.91%	N/A	N/A	N/A
StringTokenizer	100.00%	11.92%	21.30%	87.92%	0.00%	0.00%
ZipOutputStream	45.77%	100.00%	62.80%	49.48%	100.00%	66.20%
Average	73.33%	52.77%	54.69%	77.51%	53.91%	55.44%

more accurate the inferred invariants. On the other hand, utilizing raw method orderings in execution traces for specification mining is less costly. However, execution traces provide limited information of important properties that might be valuable to enhance quality of inferred FSAs. As future work, we plan to employ likely invariants in DSM to avoid incorrect merges of automaton states made by clustering algorithms.

6.3.6 Threats to Validity

There are a number of potential threats that may affect the validity of our study:

Threats to internal validity We have carefully checked our implementation, but there are errors that we did not notice. There are also potential threats related to correctness of ground truth models created by Krka et al. [46] that we used. To mitigate this threat, we have compared their models against execution traces collected from Randoop generated test cases as well as textual documentations published by library class writers (e.g., Javadocs). We revised the ground truth

Table 6.7: Precision, Recall, and F-measure: DSM with standard RNN (DSM^{RNN})

Target Library Classes	Precision	Recall	F-measure
ArrayList	57.97%	2.58%	4.95%
HashMap	100.00%	95.62%	97.76%
HashSet	71.40%	66.48%	68.86%
Hashtable	100.00%	78.48%	87.94%
LinkedList	100.00%	19.25%	32.29%
NFST	53.95%	100.00%	70.09%
Signature	48.49%	100.00%	65.31%
Socket	55.00%	48.09%	51.31%
StackAr	56.41%	98.70%	71.79%
StringTokenizer	100.00%	92.10%	95.88%
ZipOutputStream	77.55%	100.00%	87.36%
Average	74.62%	72.85%	66.69%

models accordingly.

Threats to External Validity These threats correspond to the generalizability of our empirical findings. In this work, we have analyzed 11 different library classes. This is larger than the number of target classes used to evaluate many prior studies, e.g., [46, 67, 66]. As future works, we plan to reduce this threat by analyzing more library classes to infer their automaton based specifications.

Threats to Construct Validity These threats correspond to the usage of evaluation metrics. We have followed Lo and Khoo’s approach that uses precision, recall, and F-measure to measure the accuracy of automata output by a specification mining algorithm against ground truth models [63]. Furthermore, Lo and Khoo’s approach is well known and has been adopted by many previous research works in specification mining e.g., [14, 15, 46, 26, 62, 66, 49].

6.4 Conclusion

Formal specifications are helpful for many manual software processes. In this work, we propose DSM, a new approach that employs Recurrent Neural Network Based

Table 6.8: Precision, Recall, and F-measure: DSM with GRU (DSM^{GRU})

Target Library Classes	Precision	Recall	F-measure
ArrayList	51.08%	5.17%	9.39%
HashMap	100.00%	100.00%	100.00%
HashSet	87.93%	63.70%	73.88%
Hashtable	100.00%	78.48%	87.94%
LinkedList	100.00%	21.11%	34.86%
NFST	83.54%	65.31%	73.31%
Signature	100.00%	91.22%	95.41%
Socket	53.56%	64.05%	58.34%
StackAr	67.10%	92.21%	77.67%
StringTokenizer	100.00%	100.00%	100.00%
ZipOutputStream	57.84%	100.00%	73.29%
Average	81.91%	71.02%	71.28%

Language Models (RNNML) for mining automaton based specifications. We apply Randoop, a well-known test cases generation approach, to enrich execution traces for training RNNML. From the collected traces, we construct a Prefix Tree Acceptor (PTA) and extract many features of PTA’s states. These features are then utilized by clustering algorithms to merge similar automata states and construct the final finite-state automaton. We perform our proposed approach to infer specifications for 11 target library classes. Our results show that DSM achieves an average precision, recall, and F-measure of 82.76%, 72.3%, and 71.97%, respectively. Compared to other existing specification mining algorithms, our approach outperforms the best baseline by 29.82% in terms of average F-measure.

As future work, we plan to improve DSM’s effectiveness further by integrating information of likely invariants into our deep learning based framework. Furthermore, we plan to tune DSM with many clustering algorithms aside k-means and hierarchical clustering, especially the ones that require no inputs of number of clusters (e.g., DBSCAN [31], etc.). We also plan to develop new test cases generation technique that is specialized for application of deep learning in specification mining. Finally, we plan to evaluate our approaches with more classes and

libraries in order to reduce threats to external validity.

Chapter 7

CONCLUSION AND FUTURE WORK

Software fault localization and specification mining are essential. Developers employ fault localization to find locations of faulty program elements, while specification mining approaches automatically infer formal specifications in a variety of formalism that can be used in various downstream processes (e.g., finding bugs). In this dissertation, I propose a number of hybrid based approaches for fault localization and specification mining. In terms of design paradigm, my proposed approaches combine various available models and sources of information together (i.e., AML and SpecForge) or integrate different inference/learning techniques in one framework (i.e., Savant and DSM). Summary of the proposed approaches are highlighted as follows:

1. **AML** is a multi-model fault localization approach that considers both bug reports and program spectra to localize bugs. AML also proposes the new concept of *suspicious words* to adapt *tf-idf* weighting scheme of Vector Space Model for fault localization (see Chapter 3).
2. **Savant** employs a *learning-to-rank* strategy, using *likely invariant* diffs and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure (see Chapter 4).
3. **SpecForge** synergizes many existing specification miners. The approach decomposes FSAs that are inferred by existing miners into simple constraints, through a process I refer to as *model fission*. SpecForge then filters the outlier constraints and fuses the constraints back together into a single FSA (i.e., *model fusion*) (see Chapter 5).

4. **DSM** employs deep learning, test case generation, and clustering algorithms for mining FSA based specifications from execution traces. The approach utilizes test case generation to create a rich set of training data for learning Recurrent Neural Network Language Model (RNNLM). Then, the approach constructs a Prefix Tree Acceptor (PTA) from execution traces, and extracts interesting features using the learned language model. These features are subsequently utilized by clustering algorithms for inferring several finite-state automata. Finally, DSM performs a model selection heuristic to select the most accurate FSA as the resultant FSA-based specification.

For future work, I plan to further improve AML, Savant, SpecForge, and DSM in many different ways as follows:

- **AML:** I plan to employ AML to combine many bug localization approaches including Savant. I am also interested to enhance AML's Integrator block to effectively use historical bugs for combining different bug localization models. To reduce the threats to external validity, I also plan to investigate more bug reports from additional software systems, which are implemented in different programming languages.
- **Savant:** I plan to improve *Savant* further by selectively including a subset of invariants specialized for a target buggy program version and its spectra. I also plan to include a refinement process which incrementally adds or removes invariants to produce a better ranked list of methods. Furthermore, I plan to extend my evaluation to include more bugs beyond those in the Defects4J benchmark and compare *Savant* against other fault localization approaches. I also plan to investigate the impact of number of passed and failed test cases as well as other factors on the effectiveness of *Savant*.
- **SpecForge:** I plan to improve the effectiveness of SpecForge further by increasing the number of underlying FSA miners synergized together, increas-

ing the number of constraint templates, and developing an approach that can infer good configurations of constraint templates and selection heuristics based on training data. I also plan to reduce the threats to external validity further by experimenting with additional target library classes and execution traces. Moreover, I plan to extend SpecForge to mine parametric specification following the work by Lee et al. [54] and Lo et al. [25, 68]. Finally, I plan to combine many more specification miners including DSM to infer more accurate specifications.

- **DSM:** I plan to improve DSM’s effectiveness further by integrating information of likely invariants into our deep learning based framework. Furthermore, I plan to improve DSM by considering many clustering algorithms aside from k-means and hierarchical clustering, especially the ones that require no inputs of number of clusters (e.g., DBSCAN [31], etc.). I also plan to develop new deep learning based solution that leverages likely invariants inferred by Daikon for mining FSA-based specifications. Finally, I plan to evaluate DSM with more classes and libraries in order to reduce threats to external validity.

My long-term goal is to further improve the effectiveness and efficiency of Savant as well as AML to be adopted in practice by developers to find bugs. I am also particularly interested to combine fault localization and specification techniques together. In fact, fault localization tools can utilize specifications to detect violated properties in execution traces or in program implementations.

BIBLIOGRAPHY

- [1] Apache Ant. <http://ant.apache.org/>. Accessed: 2015-07-15.
- [2] Apache Lucene. <http://lucene.apache.org/core/>. Accessed: 2015-07-15.
- [3] AspectJ. <http://eclipse.org/aspectj/>. Accessed: 2015-07-15.
- [4] Mysql 5.6 full-text stopwords. <http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html>. Accessed: 2015-07-15.
- [5] Rhino. <http://developer.mozilla.org/en-US/docs/Rhino>. Accessed: 2015-07-15.
- [6] R. Abreu, P. Zoetewey, R. Golsteijn, and A.J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.
- [7] Rui Abreu, Peter Zoetewey, and Arjan J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*, 2007.
- [8] Robert Andrews, Joachim Diederich, and Alan B. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowl.-Based Syst.*, 8(6):373–389, 1995.
- [9] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*, pages 35–39, 2005.
- [10] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.
- [11] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE Int. Conference on Software Engineering, ICSE '10*, 2010.
- [12] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.
- [13] Ivan Beschastnikh. *Modeling systems from logs of their behavior*. PhD thesis, 2013.

- [14] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Software Eng.*, 41(4):408–428, 2015.
- [15] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011.
- [16] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 100(6):592–597, 1972.
- [17] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [18] Junyoung Chung, Çağlar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.
- [19] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [20] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.
- [21] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 342–351, New York, NY, USA, 2005. ACM.
- [22] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. Automatically generating test cases for specification mining. *IEEE Trans. Software Eng.*, 38(2):243–257, 2012.
- [23] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSA 2010, Trento, Italy, July 12-16, 2010*, pages 85–96, 2010.
- [24] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.
- [25] David Lo, Ganesan Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Mining quantified temporal rules: Formalism, algorithms, and

- evaluation. In *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pages 62–71, 2009.
- [26] David Lo and Siau-Cheng Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 265–275, 2006.
- [27] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Automated abstractions for contract validation. *IEEE Trans. Software Eng.*, 38(1):141–162, 2012.
- [28] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.
- [29] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE’ 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420, 1999.
- [30] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
- [31] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996.
- [32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise.
- [33] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.
- [34] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6, 000 projects: lightweight cross-project anomaly detection. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSA 2010*, pages 119–130, 2010.

- [35] J. Han and M. Kamber. *Data Mining: Concepts and Techniques* (2nd ed.). Morgan Kaufmann, 2006.
- [36] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
- [37] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [38] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, May 1997.
- [39] T. Joachims. Svm^{rank}: Support vector machine for ranking. www.cs.cornell.edu/people/tj/svm_light/svm_rank.html. Accessed: 2015-07-15.
- [40] J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM international Conference on Automated software engineering*, 2005.
- [41] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Int. Symposium on Software Testing and Analysis, ISSTA '14*, pages 437–440, 2014.
- [42] Sunghun Kim and E. James Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pages 173–174, 2006.
- [43] John C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Lus G. Nakano. Why are formal methods not used more widely? In *Fourth NASA Formal Methods Workshop*, pages 1–12, 1997.
- [44] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners’ expectations on automated fault localization. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis*. ACM, 2016.
- [45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114, 2012.
- [46] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pages 178–189, 2014.

- [47] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 115–125, 2015.
- [48] Tien-Duy B Le and David Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 331–340. IEEE, 2015.
- [49] Tien-Duy B. Le and David Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 331–340, 2015.
- [50] Tien Duy B. Le, Shaowei Wang, and David Lo. Multi-abstraction concern localization. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 364–367, 2013.
- [51] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
- [52] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [53] Ching-Pei Lee and Chuan-bi Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.
- [54] Choonghwan Lee, Feng Chen, and Grigore Rosu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 591–600, 2011.
- [55] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92. IEEE, 2015.
- [56] Wenchao Li, Alessandro Forin, and Sanjit A Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference*, pages 755–760. ACM, 2010.

- [57] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
- [58] Ben Liblit, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 141–154, 2003.
- [59] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.
- [60] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. Sober: Statistical model-based bug localization. In *ESEC/FSE*, 2005.
- [61] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 234–243, 2007.
- [62] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 557–566. ACM, 2009.
- [63] David Lo and Siau-Cheng Khoo. QUARK: empirical assessment of automaton-based specification miners. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 51–60, 2006.
- [64] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance*, 20(4), 2008.
- [65] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. *Autom. Softw. Eng.*, 19(4), 2012.
- [66] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 345–354, 2009.
- [67] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063–2076, 2012.

- [68] David Lo, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. *Sci. Comput. Program.*, 2012.
- [69] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2016.
- [70] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 501–510, 2008.
- [71] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.
- [72] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. Are faults localizable? In *MSR*, pages 74–77, 2012.
- [73] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972–990, 2010.
- [74] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.
- [75] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [76] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 117–126, 2008.
- [77] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Software Eng.*, 37(4):486–508, 2011.
- [78] Weikai Miao and Shaoying Liu. A formal specification-based integration testing approach. In *SOFL*, pages 26–43, 2012.
- [79] Tomas Mikolov, Anoop Deoras, Stefan Kombrink, Lukás Burget, and Jan Cernocký. Empirical evaluation and combination of advanced language modeling techniques. In *INTERSPEECH 2011, 12th Annual Conference of the*

International Speech Communication Association, Florence, Italy, August 27-31, 2011, pages 605–608, 2011.

- [80] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.
- [81] Anders Møller. dk.brics.automaton — Finite-state automata and regular expressions for Java. <http://www.brics.dk/automaton/>, 2010.
- [82] Mozilla. Bug fields. <https://bugzilla.mozilla.org/page.cgi?id=fields.html>. Accessed: 2015-03-16.
- [83] K.P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
- [84] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.
- [85] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM Int. Conference on Automated Software Engineering*, 2011.
- [86] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13*, pages 772–781, 2013.
- [87] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 19–30. ACM, 2014.
- [88] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada*, pages 199–209, 2011.
- [89] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 609–620, 2017.

- [90] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, Providence, RI, USA, 1977. DOI: 10.1109/SFCS.1977.32.
- [91] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [92] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.
- [93] Michael Pradel, Philipp Bichsel, and Thomas R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010.
- [94] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. *Proceedings of Workshop on Automated and Algorithmic Debugging*, 2003.
- [95] Shivani Rao and Avinash C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 43–52, 2011.
- [96] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 30–39, 2003.
- [97] Mark Roberts and Michael Ernst. Personal Emails.
- [98] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*, pages 23–32, Lawrence, KS, USA, November 2011.
- [99] Lior Rokach and Oded Maimon. Clustering methods. In *The Data Mining and Knowledge Discovery Handbook.*, pages 321–352. 2005.
- [100] Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.

- [101] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 345–355, 2013.
- [102] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram S. Adve. Using likely invariants for automated software fault localization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013.
- [103] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.
- [104] Bunyamin Sisman and Avinash C. Kak. Incorporating version histories in information retrieval based bug localization. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 50–59, 2012.
- [105] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.
- [106] G. Tasse. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
- [107] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pages 271–280, 2012.
- [108] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L’Aquila, Italy*, pages 248–257, 2008.
- [109] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 301–310, 2013.
- [110] S. Wang and D. Lo. History, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, 2014.

- [111] Shaowei Wang, David Lo, and Julia Lawall. Compositional vector space models for improved bug localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 171–180, 2014.
- [112] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 297–308, 2016.
- [113] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 87–98, 2016.
- [114] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.
- [115] Xin Xia, Xiaozhen Zhou, David Lo, and Xiaoqiong Zhao. An empirical study of bugs in software build systems. In *2013 13th International Conference on Quality Software, Naging, China, July 29-30, 2013*, pages 200–203, 2013.
- [116] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31, 2013.
- [117] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In *5th Int. Symposium on Search Based Software Engineering - SSBSE 2013, Proceedings*, pages 224–238, 2013.
- [118] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *30th IEEE Int. Conference on Software Maintenance and Evolution, 2014*, pages 191–200, 2014.
- [119] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *ICSE*, May 2006.
- [120] Xin Ye, Razvan C. Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 689–699, 2014.
- [121] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *4th International Symposium Search Based Software Engineering SSBSE*, 2012.

- [122] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 2002.
- [123] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, 1999.
- [124] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.
- [125] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
- [126] Hao Zhong and Zhendong Su. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 803–816, 2013.
- [127] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 14–24, 2012.