# Effective Program Analyses for Automated Software Testing and Error Diagnosis

Sai Zhang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Michael D. Ernst, Chair

Dan Grossman

Luke Zettlemoyer

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

**Abstract**

Effective Program Analyses for Automated Software Testing
and Error Diagnosis

Sai Zhang

Chair of the Supervisory Committee:
Professor Michael D. Ernst
Department of Computer Science and Engineering

This dissertation presents five program analysis techniques for improving automated software testing and error diagnosis. Two techniques aim to help software developers build reliable software and three techniques assist software users in quickly identifying the root cause of an exhibited software error.

The first two techniques help software developers validate software's behaviors before delivering it to users. The first technique, called Palus, assists developers in creating unit test suites for software systems with constrained interfaces. Palus combines dynamic model inference, static analysis, and random testing to automatically generate unit tests that are legal and behaviorally-diverse. Tests generated by Palus achieve higher code coverage, and detect more bugs than tests generated by existing techniques. When a test fails, a developer must understand the cause of the failure before fixing the bug. The second technique, called FailureDoc, helps developers interpret the test results. FailureDoc uses static analysis, runtime monitoring, and statistical reasoning to augment a failed test with contextually-relevant debugging clues: code comments that provide useful facts about the failure, and assists developers in understanding why a test fails.

The other three techniques help software users diagnose exhibited software errors. The third technique, called ConfDiagnoser, assists software users in troubleshooting configuration errors. It uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options whose values should be changed. The fourth technique, called

ConfSuggester, extends ConfDiagnoser to support configuration error diagnosis for evolving software systems. ConfSuggester uses the desired behavior of the old software version as a baseline against which to compare new program behavior, and reasons about the root-cause configuration option. The fifth technique, called FlowFixer, helps users repair UI workflow errors. Given a new software version's changed UI, in which a user's desired action is not possible, FlowFixer uses dynamic profiling, static analysis, and random testing to inform users of replacement UI actions to fix a broken workflow.

These techniques have been implemented and evaluated on real-world software systems. The experimental results and user studies have shown that the implemented techniques are accurate, efficient, and practically useful.

# TABLE OF CONTENTS

Page

List of Figures . . . . . v

Chapter 1: Introduction . . . . . 1
1.1 Motivation . . . . . 1
1.1.1 Improving Software Reliability via Automated Testing . . . . . 1
1.1.2 Performing Error Diagnosis by Software Users . . . . . 2
1.2 Thesis and Contributions . . . . . 3
1.3 Challenges and Insights . . . . . 5
1.4 Scope . . . . . 10
1.5 Outline . . . . . 10

Chapter 2: Generating Tests for Software with Constrained Interfaces . . . . . 13
2.1 Problem . . . . . 13
2.2 Example . . . . . 15
2.3 Technique . . . . . 18
2.3.1 Dynamic Analysis: Model Inference from Sample Executions . . . . . 19
2.3.2 Static Analysis: Model Expansion with Dependence Analysis . . . . . 22
2.3.3 Guided Test Generation . . . . . 24
2.4 Tool Implementation . . . . . 27
2.5 Evaluation . . . . . 28
2.5.1 Research Questions . . . . . 28
2.5.2 Subject Programs and Tools . . . . . 28
2.5.3 Experimental Procedure . . . . . 29
2.5.4 Coverage Results . . . . . 30
2.5.5 Bug Finding Ability . . . . . 35
2.5.6 Experiment Discussion and Conclusion . . . . . 38
2.6 Summary . . . . . 39

Chapter 3: Explaining Test Failures with Documentation Inference . . . . . 41
3.1 Problem . . . . . 41
3.2 Technique . . . . . 44
3.2.1 Value Replacement . . . . . 45
3.2.2 Execution Observation . . . . . 48
3.2.3 Failure Correlation . . . . . 50
3.2.4 Documentation Generation . . . . . 52
3.3 Evaluation . . . . . 54
3.3.1 Experiment: Inferring Debugging Clues . . . . . 55
3.3.2 User Study: Understanding Failed Tests . . . . . 59
3.4 Summary . . . . . 62

Chapter 4: Diagnosing Configuration Errors in Complex Software Systems . . . . . 63
4.1 Problem . . . . . 63
4.2 Example . . . . . 64
4.3 Technique . . . . . 66
4.3.1 Overview . . . . . 67
4.3.2 Configuration Propagation Analysis . . . . . 68
4.3.3 Configuration Behavior Profiling . . . . . 69
4.3.4 Configuration Deviation Analysis . . . . . 70
4.3.5 Discussion . . . . . 73
4.4 Tool Implementation . . . . . 73
4.5 Evaluation . . . . . 74
4.5.1 Subject Programs . . . . . 74
4.5.2 Evaluation Procedure . . . . . 75
4.5.3 Results . . . . . 76
4.5.4 Discussion . . . . . 83
4.6 Summary . . . . . 84

Chapter 5: Diagnosing Configuration Errors in Evolving Software Systems . . . . . 85
5.1 Problem . . . . . 85
5.2 An Empirical Study of Real-World Configuration Changes . . . . . 86
5.2.1 Subject Programs and Study Methodology . . . . . 86
5.2.2 Findings . . . . . 88
5.2.3 Threats to Validity . . . . . 88
5.3 Technique . . . . . 89

5.3.1 Overview . . . 89
5.3.2 Instrumentation and Demonstration . . . 90
5.3.3 Execution Trace Comparison . . . 91
5.3.4 Configuration Option Recommendation . . . 95
5.3.5 Discussion . . . 98
5.4 Tool Implementation . . . 99
5.5 Evaluation . . . 99
5.5.1 Subject Programs . . . 99
5.5.2 Evaluation Procedure . . . 100
5.5.3 Results . . . 102
5.5.4 Discussion . . . 107
5.6 Summary . . . 107

Chapter 6: Fixing Broken Workflows for Evolving GUI Applications . . . 108
6.1 Problem . . . 108
6.2 Technique . . . 110
6.2.1 Overview . . . 110
6.2.2 Profiling a Broken Workflow . . . 111
6.2.3 Matching Methods across Versions . . . 113
6.2.4 Executing Random UI Actions . . . 114
6.2.5 Recommending Replacement UI Actions . . . 117
6.2.6 Discussion . . . 118
6.3 Tool Implementation . . . 119
6.4 Evaluation . . . 120
6.4.1 Subject Programs and Broken Workflows . . . 120
6.4.2 Evaluation Procedure . . . 122
6.4.3 Results . . . 122
6.4.4 Discussion . . . 130
6.5 Summary . . . 131

Chapter 7: Related Work . . . 132
7.1 Program Specification Inference . . . 132
7.1.1 Dynamic Inference . . . 132
7.1.2 Static Inference . . . 133
7.1.3 Statistical Inference . . . 135
7.2 Automated Software Testing . . . 136

7.2.1 Test Generation . . . 136
7.2.2 Testing Configurable Software . . . 139
7.2.3 Testing GUI Applications . . . 140
7.3 Software Error Diagnosis and Patching . . . 141
7.3.1 Automated Debugging Techniques . . . 142
7.3.2 Diagnosis of Configuration Errors . . . 148
7.3.3 Software Failure Explanation . . . 149
7.3.4 Patching Software Errors . . . 151

Chapter 8: Future Work . . . 153
8.1 Developing User Interfaces for Program Analysis Tools . . . 153
8.2 Conducting User Studies to Understand Analysis Usability . . . 154
8.3 Improving Non-Functional Software Properties . . . 154
8.4 Classifying Software Errors . . . 155
8.5 Recovering from Software Failures . . . 156
8.6 Analyzing Non-Executable Artifacts . . . 157
8.7 Preventing and Resolving End-user Errors . . . 158
8.8 Automating End-user Programming . . . 159

Chapter 9: Conclusion . . . 160
9.1 Lessons Learned . . . 161

Bibliography . . . 166

## LIST OF FIGURES


Figure Number — Page

2.1 Three classes taken from the tinySQL project [195]. . . . 16
2.2 A sample method call sequence for the code in Figure 2.1. . . . 16
2.3 The call sequence model built by a dynamic-random approach (Palulu [10]) for classes in Figure 2.1. . . . 17
2.4 A testing oracle expressed as a JUnit theory. . . . 19
2.5 Architecture of the Palus tool. . . . 19
2.6 An enhanced call sequence model with direct state transition dependence edges for the example in Figure 2.1. . . . 21
2.7 Sequence generation algorithm in Palus. . . . 24
2.8 Guided sequence generation algorithm in Palus. . . . 25
2.9 Structural coverage of tests generated by Randoop, Palulu, RecGen, and Palus. . . . 31
2.10 Breakdown of the line coverage increase for 6 subjects in table 2.1. . . . 33
2.11 Model size (node number in log scale) and test coverage after applying the *kTail* algorithm with $k = 3$. . . . 35

3.1 A human-written failed test that reveals a potential error in the JDK. . . . 42
3.2 An automatically-generated failed test that reveals an error in JDK version 1.6. . . . 42
3.3 The failing test of Figure 3.1 with code comments inferred by the FailureDoc tool. . . 44
3.4 The failing test of Figure 3.2 with code comments inferred by the FailureDoc tool. . . 44
3.5 Illustration of FailureDoc's workflow with a simple example. . . . 45
3.6 Algorithm for isolating a set of suspicious statements. . . . 52
3.7 Subject programs and experimental results used in evaluating FailureDoc. . . . 53
3.8 A failed test in Apache Commons Collections. . . . 56
3.9 FailureDoc user study results. . . . 58

4.1 The top-ranked configuration option in ConfDiagnoser's error report for the motivating example in Section 4.2. . . . 65
4.2 Simplified code excerpt from Randoop [148] corresponding to the configuration problem reported in Figure 4.1. . . . 66
4.3 Workflow of the ConfDiagnoser technique. . . . 68
4.4 Subject programs used in evaluating ConfDiagnoser. . . . 75

4.5 The 14 configuration errors used in evaluating ConfDiagnoser. . . . . 75
4.6 Experimental results in diagnosing software configuration errors. . . . . 77
4.7 ConfDiagnoser's performance. . . . . 79
4.8 Comparison with different execution profile selection strategies (Section 4.5.3). . . 82

5.1 A list of studied open-source software systems and their characteristics. . . . . 86
5.2 The total number of new, deleted, and modified configuration options for each subject program. . . . . 87
5.3 Types of configuration changes identified in our study from the subject programs in Figure 5.1. . . . . 87
5.4 The number of configuration changes of each type described in Figure 5.3. . . . . 88
5.5 The architecture of our ConfSuggester technique. . . . . 89
5.6 Algorithm for matching statements from two methods in ConfSuggester. . . . . 92
5.7 The deviation function used in ConfSuggester. . . . . 96
5.8 Algorithm for recommending configuration options in ConfSuggester. . . . . 97
5.9 All subject programs used in evaluating ConfSuggester. . . . . 100
5.10 All configuration errors used in evaluating ConfSuggester and the experimental results. 101
5.11 ConfSuggester's performance. . . . . 103
5.12 Experimental results of evaluating two design choices of ConfSuggester. . . . . 105

6.1 An example broken workflow in the Crossword [37] program. . . . . 109
6.2 Illustration of FlowFixer's 4 steps in repairing a broken workflow using the Crossword example from Figure 6.1. . . . . 112
6.3 Algorithm for randomly executing UI actions on a GUI application. . . . . 114
6.4 Algorithm for recommending replacement UI actions for a broken workflow. . . . 115
6.5 Subject programs and broken workflows used to evaluate FlowFixer. . . . . 121
6.6 Experimental results in repairing broken workflows. . . . . 123
6.7 An example broken work in the Gantt Project program. . . . . 124
6.8 Number of broken workflows that FlowFixer can repair using the three heuristics in Section 6.2.3. . . . . 124
6.9 Experimental results in executing random UI actions. . . . . 125
6.10 FlowFixer's performance in repairing broken workflows. . . . . 127
6.11 Experimental results in comparing FlowFixer with a REST-based technique [67]. . 128
6.12 Comparison of FlowFixer with a static analysis-based approach described in Section 6.4.3. . . . . 129

## ACKNOWLEDGMENTS

Portions of this dissertation were previously published at ISSTA 2011 [236] (Chapter 2), ASE 2011 [238] (Chapter 3), ICSE 2013 [232] and ICSE 2014 [233] (Chapter 4), and ISSTA 2013 [235] (Chapter 6). The material included in this dissertation has been updated and extended.

I greatly enjoyed the five years I spent working on my Ph.D. degree at University of Washington. I had exciting research topics, excellent advisors, talented collaborators, and great friends. I am grateful to many people who have contributed to this dissertation with their support, collaboration, mentorship, and friendship. I apologize in advance to those who I did not explicitly mention in this dissertation. You all have my gratitude for your help.

First of all, I would like to thank my advisor, Michael Ernst, for supporting me with freedom and guidance throughout my Ph.D. studies. Mike has provided me valuable technical and professional advice, which will continue to benefit me in my future career. Mike has also kept me focused on my goals and made certain that I never needed to worry about funding. His wisdom and passion for research have been a continuous inspiration to me.

I would like to thank Dan Grossman, Darko Marinov, and Luke Zettlemoyer for all their help during my graduate studies. They served on my thesis committee and helped me improve the writing of this dissertation. They also helped me with their advice and discussions on doing research. I wish I had time to collaborate more with them. I also thank Jim Pfaendtner for serving as a Graduate School Representative (GSR) on my thesis committee.

David Notkin was a key mentor for me. He served on my Ph.D. qualifying and general exam committee and provided valuable advice on several matters. Rest in peace, David.

Over the years, I have worked on a number of other projects that are not included in this dissertation. All of these projects were collaborations with other researchers — Yuriy Brun, Ivan Beschastnikh, Darioush Jalali, Wing Lam, Hao Lü, Kıvanç Muşlu, David Notkin, Roykrong Sukkerd, Yuyin Sun, Jochen Wuttke, Congle Zhang, Cheng Zhang, and Jianjun Zhao. The perspectives and

humor of all of these people have made my graduate experience memorable, productive, and fun.

Many colleagues kindly offered their help in shaping and improving my research. I want to especially thank Jamie Andrews, Shay Artzi, Ivan Beschastnikh, Yuriy Brun, Brian Burg, Tom Bergan, Yingyi Bu, Werner Dietl, Pedro Domingos, Sebastian Elbaum, Michael Ernst, James Fogarty, Colin Gordon, Dan Grossman, René Just, Darioush Jalali, Wing Lam, Mark Harman, Kaituo Li, Qingzhou Luo, Hao Lü, Kıvanç Muşlu, Darko Marinov, Gail Murphy, David Notkin, Carlos Pacheco, Barbara Ryder, David Saff, Todd Schiller, Dan Suciu, Roykrong Sukkerd, Yuyin Sun, Zachary Tatlock, Frank Tip, Mandana Vaziri, Jochen Wuttke, Xusheng Xiao, Tao Xie, Guoqing Xu, Andreas Zeller, Lingming Zhang, Congle Zhang, Cheng Zhang, and Jianjun Zhao.

Looking further back, my mentor, Jianjun Zhao, from Shanghai Jiao Tong University encouraged and inspired me to study Computer Science and attend graduate school. He was instrumental in getting me started with Software Engineering research.

Many thanks to my friends, Yingyi Bu, Weitao Chen, Shumo Chu, Peng Dai, Julia Deng, Hao Du, Jieling Han, Yanping Huang, Yang Li, Lily Lee, Xiao Ling, Hao Lü, Yi Jin, Xu Miao, Hoifung Poon, Qifan Pu, Qi Shan, You Ren, Haichen Shen, Yuyin Sun, Kallie Tang, Edward Wu, Jingjing Wang, Xiao Wang, Changchang Wu, Zhe Xu, Shengliang Xu, Shulin Yang, Jijiang Yan, Yeqin Zeng, and Congle Zhang, who share their joy and bear my depression, which contributed to my five colorful years at UW.

Last but not least, I am grateful to my family. My parents and parents-in-law have encouraged me and believed in me for many years. My wife gives me continuous support all the time.

## DEDICATION

to my family

# Chapter 1

# INTRODUCTION

Billions of people rely on software, from communications and transportation to entertainment and healthcare. As our dependence on software grows, so does the need to make software systems more reliable. As the complexity of modern software increases, users may struggle to discover software features, understand software behaviors, and resolve software usage problems.

This dissertation focuses on developing practical program analyses for improving software reliability (via automated testing) and end-user error diagnosis with minimal manual input and without formal specifications. The developed program analyses infer software behavioral models and properties from execution traces to guide automated testing, and correlate software behavioral differences (captured by the inferred models and properties) to the root causes in error diagnosis.

This chapter describes the motivation of this dissertation (Section 1.1), summarizes the contributions (Section 1.2), discusses the challenges and a high-level overview of the insights exploited in this dissertation (Section 1.3), refines the scope (Section 1.4), and gives an organization of the remainder of the dissertation (Section 1.5).

## *1.1 Motivation*

The research presented in this dissertation is motivated by the need to improve the reliability of modern software (Section 1.1.1) and the need to perform end-user error diagnosis (Section 1.1.2).

### *1.1.1 Improving Software Reliability via Automated Testing*

Software developers must validate software's behaviors before delivering it to users. Testing is a widely used technique to validate software behaviors. Studies show that testing accounts for more than half of the total software development cost [193, 200].

Automated testing can significantly help developers develop and maintain reliable software. However, test automation is mainly limited to test execution, while test creation and comprehension

remain manual and mostly ad hoc. Existing test generation tools [32, 106, 148, 217] often cannot effectively generate satisfactory tests to achieve good code coverage or expose defects for complex software systems, such as software with constrained interfaces[1]. In addition, when software exhibits unexpected behaviors (i.e., test failures) during automated testing, developers have long struggled with understanding their root causes. Few techniques exist to explain why a test fails. Understanding why a test failure arises or even knowing which part of the test code should be inspected first in debugging is a non-trivial task. This is a particular problem for automatically-generated tests which are often long and have poor readability, but it is also relevant for human-written tests.

### *1.1.2 Performing Error Diagnosis by Software Users*

Software users need to diagnose and resolve software usage errors. However, performing error diagnosis is challenging for software users. Unlike developers, most software users (including system administrators) may have little or no programming knowledge and cannot access (much less understand) the source code. For example, software configuration errors are errors in which the software code and the input are correct, but the software is misconfigured so that it does not behave as desired by the user. Such errors are prevalent and can lead the software to crash, produce erroneous output, or simply perform poorly. Although configuration errors are prevalent and severe, they are actionable for ordinary software users to fix — users can fix a configuration error by simply changing the misconfigured options to the desired values. However, few techniques and tools exist to directly address such end-user-fixable errors. Most of the existing error diagnosis tools are designed for software developers: they require access to the source code [159], or need OS-level support [184], or assume the person using the tool can answer difficult questions like, "is the software current working?" or, "why is the software not working?" by writing a machine-checkable testing oracle [209].

We believe that many such errors could be effectively diagnosed by software users if users are supplied with practical diagnosis tools, analogous to tools used by developers for software debugging. A practical error diagnosis tool for software users should eliminate unrealistic assumptions (e.g., the need to write a testing oracle) and provide actionable solutions to resolve the encountered error (e.g., informing which specific configuration options should a user change).

---

[1] Testing software with constrained interfaces, as opposed to testing general utility libraries, requires invoking method calls in a specific order with specific arguments

### *1.2 Thesis and Contributions*

This dissertation demonstrates the following principal thesis statement:

> Automated software testing and end-user error diagnosis can be performed with minimal manual input and without formal specifications, by inferring software models and properties from execution traces and correlating software behavioral differences (captured by the inferred models and properties) to the root causes.

This dissertation makes the following contributions:

- **An automated test generation technique for software with constrained interfaces.** We present Palus, an automated test generation technique. Palus is specifically designed for software with constrained interfaces. It combines dynamic model inference, static analysis, and random testing, to create tests that are legal and behaviorally-diverse. Palus exploits three key insights. First, it uses dynamic analysis to infer a call sequence model from a correct sample execution. The inferred call sequence model contains both method-call constraints (e.g., `connectToDatabase`() is called before `queryDatabase(sqlQuery)`) and argument value constraints (e.g., the value of `sqlQuery` is a valid SQL statement). Second, Palus uses static analysis to explore different program behaviors that are not covered by the dynamic analysis. Third, guided by the results of both static and dynamic analyses, Palus uses a randomized algorithm to diversify a specific legal method-call sequence to create legal and behaviorally-diverse tests.

- **An automated test failure explanation technique.** Palus helps developers create tests, but it does not help developers interpret the test results. To address this problem, this dissertation presents FailureDoc, a technique to explain why a test fails. FailureDoc differs from existing techniques in that it augments a failed test with contextually-relevant debugging clues: code comments that provide useful facts about the failure, such as "the test fails because variable `obj` does not implement the `Comparable` interface". FailureDoc simulates developers' debugging activity, and automates the following common debugging process: given a failed test, a developer often makes some (minimal) edits to make it pass, observes the difference between

passing and failing executions, then generalizes those failure-correcting edits to understand the failure cause. FailureDoc draws on techniques from randomized algorithms, statistical inference, and dynamic invariant detection. It repeatedly mutates the source code of a failed test, observes the behaviors of the mutated test, generalizes observations, and translates observations into explanatory code comments.

- **Two automated configuration error diagnosis techniques.** This dissertation presents two techniques, ConfDiagnoser and ConfSuggester, to assist software users in finding the root-cause configuration options. ConfDiagnoser diagnoses configuration errors in a single software version, and ConfSuggester diagnoses configuration errors introduced during software evolution by working on two different versions of the same software.

  ConfDiagnoser exploits two key insights. First, a program's control flow, rather than data flow, propagates the majority of the effects of a configuration option. Second, the control flow differences between a correct execution and an erroneous execution can approximate the program behavioral differences and provide evidence for which parts of the program might be behaving abnormally and why. ConfDiagnoser uses dynamic analysis to monitor a program's execution and compares an erroneous execution with a set of correct executions (obtained by running tests or examples from software user manual) to identify the control flow differences. Then, ConfDiagnoser uses static analysis to identify the configuration options responsible for each control flow difference as the likely root causes.

  ConfSuggester extends ConfDiagnoser in two ways. First, ConfSuggester is cognizant of software evolution and works on two different versions of the same software. It uses the desired behavior on the old software version as a baseline with which to compare new program behavior against, and only reasons about the behavioral differences. Second, when comparing the undesired execution trace with a correct execution trace, ConfSuggester not only uses a predicate's deviation value (used in ConfDiagnoser and defined in Chapter 4) to reason about the most suspicious options, but also uses the statements controlled by a predicate's evaluation result as an indicator to decide the root-cause options.

- **An automated broken workflow repair technique.** GUI evolution can cause usability prob-

lems for software users by breaking an existing, well-established workflow — a sequence of UI actions to complete a specific task — on an old software version. Such broken workflows are another type of software usage error that a user can fix. This dissertation presents a program-analysis-based technique, called FlowFixer, to fix broken workflows. FlowFixer first uses a dynamic analysis to record the methods invoked by each workflow. Then, it uses a static analysis to match each invoked method to its corresponding, possibly updated method in the new software version. Finally, FlowFixer uses random testing to execute each UI action on the new software version, and reasons about the most likely replacement UI actions based on the observed effects. FlowFixer informs end-users of specific actions to fix a broken workflow.

- **Implementation of program analysis tools.** We describe the design and implementation of each program analysis. Our implementations support Java software. For each analysis, we present its implementation details and discuss some possible alternative implementations.

- **Empirical evaluations.** We describe a series of experiments and user studies to show that the developed techniques are scalable to real-world programs, run in a practical amount of time, and the results are useful for software developers to improve software reliability and for software users to diagnose software errors.

### *1.3 Challenges and Insights*

We faced several challenges when we started this work:

1. **Program specifications are often absent.** Program specifications are rarely written down, and so many software systems are completely lacking in formal or informal specifications and other documentation. The lack of program specification brings great challenges to automated testing and error diagnosis. Take automated test generation for object-oriented software as an example. A unit test for an object-oriented program often consists of a sequence of method calls that create and mutate objects, then use them as arguments to a method under test. Many software systems have constrained interfaces, and correct operations require method calls to occur in a certain order with specific arguments. Such implicit specifications must be satisfied when a test generator creates new tests. For example, when testing a database system, a test

must call `connectToDatabase()` before calling `queryDatabase(sqlQuery)`, and the value of the `sqlQuery` argument must be a valid SQL statement. Otherwise, the test code will be illegal and useless. Without a specification, a test generator has almost no guidance in generating legal tests. Further, lacking program specifications also makes automated error diagnosis more difficult. When an error arises, it is difficult for an analysis tool to identify the violated assumptions upon which the program's correct behavior depends. A tool can easily miss inspecting or checking program states, input values, or execution conditions that are relevant to the error.

2. **Limited information is available when a software failure occurs.** A software failure (e.g., a failed test) indicates a potential bug in the code. A developer must understand the cause of the failure and confirm its validity before starting bug-fixing. A core dump (or a stacktrace) gives the information about the final program state. Developers must manually connect the available failure symptom with possible failure causes. Suppose the developer finds an illegal value in some variable: How did that value come to be? The input that caused the failure may have long been overwritten by later input: What was it that caused the problem in the first place? Which part of the test code should be inspected first? We give concrete examples in Chapter 3.

3. **Explaining software failures requires contextually-relevant information.** Most existing tools approximate the failure explanation process by localizing the likely buggy program fragments, such as a set of suspicious statements [102]. The statements isolated by existing tools [86, 102] give limited information for understanding the failure: they may only indicate which part of the program is relevant, but do not tell why the program fails. Understanding the root cause of a failure typically involves complex activities, such as navigating program dependencies and re-running the program with different inputs. Simply giving the suspicious program statements, as most existing approaches do, is not enough for the developers to understand the failure. This exposes two key challenges for designing failure explanation techniques: what additional information should be provided to developers? And how should a program analysis derive additional information from the limited available information in a

software failure?

4. **It is hard for users to specify the desired behavior of a software system.** When suspicious software behavior exhibits, it is often challenging for users (or even developers) to answer difficult (but critical) questions like, "is the software currently working?" or, "why is the software not working?" by specifying a machine-checkable testing oracle to verify the suspicious behavior. Most software users can clearly describe what the task is, but they are often stuck with the process of how to communicate their intentions to an error diagnosis tool. Thus, non-expert users need a tool that can naturally describe their needs and connect their intentions to actionable solutions. We give concrete examples in Chapters 4 and 5.

5. **Most software users have little or no programming knowledge for error diagnosis.** When developers diagnose a software error, they can use interactive debuggers to reproduce the error, observe the execution, and identify the possible solutions. However, software users usually treat the whole software system as a black box (even when the source code is available). General programming languages and programming tools have never been easy for software users who are not professional programmers. Learning or understanding a practical programming language or a programming tool often requires a substantial amount of time and energy that a typical user would not prefer to invest. Thus, a practical program analysis technique for diagnosing end-user errors should not expect users to use developer tools; or write code, program specifications, and annotations of any form.

To address these challenges, this dissertation exploits the following insights when designing program analysis techniques. Each insight addresses a corresponding challenge listed above.

1. **Mining likely program specifications from correct execution traces.** To address the challenge of obtaining program specifications from undocumented programs, our techniques mine likely partial program specifications from correct execution traces. The correct execution traces can be obtained by running existing test suites or usage examples as described in a software system's user manual. Our techniques instrument the target program to trace certain program elements (such as, methods, instructions, and variables), run the instrumented program over a

test suite or some usage examples, and infer program properties that held or did not hold over the instrumented program elements as partial specifications. The inferred program properties include legal method-call orders (used in the Palus test generator described in Chapter 2), variable values (used in the FailureDoc explanation tool described in Chapter 3), predicate behaviors (used in the ConfDiagnoser and ConfSuggester error diagnosis tools described in Chapter 4 and Chapter 5, respectively), and method invocations (used in the FlowFixer tool described in Chapter 6).

Although program specifications mined from dynamic execution traces are not complete or sound, they are useful, which is an arguably more important property. This characterization fits naturally a testing or error diagnosis tool, since the output of such tools are often inspected by humans. Humans can always inspect and check the tool output, ignoring or investigating results that do not make sense. Moreover, an imperfect result may put them on the track or suggest valuable information to them, even if it is not exactly correct or complete. The dynamically-inferred (partial) specifications are also complementary to static analyses — they can strengthen the weaknesses of static analysis while static analysis covers for its deficiencies. For example, the Palus automated test generator described in Chapter 2 combines both dynamic and static approaches. It first uses a dynamic analysis to observe concrete executions and infers software behavioral models, then employs a static analysis to examine program code and reasons over all possible behaviors that might arise at run time to make the dynamically-inferred model more complete.

2. **Leveraging automated test generation to gain additional failure-relevant observations.** To address the challenge of obtaining useful information from a single failing execution (e.g., from a failed test), two techniques in this dissertation employ automated test generation tools to create additional tests as needed, execute the generated tests, and observe their outcomes. The aim of test generation is to gain additional failure-relevant information by producing tests that (1) are similar to the failing test, but (2) differ with respect to specific individual facts. The general underlying motivation for this insight is that we want an actual cause to be correlated with the failure occurrence: if the cause is present, the failure is present; if the cause is absent, so is the failure. For a certain program property, if there are both a passing and a failing test that

differ with respect to that property, but possibly with respect to others as well, this is a good indicator that this property may not be essential to the failure. Therefore, our techniques do not try to logically deduce the failure cause, but rather infer a strong correlation for which we aim to find helpful properties, such as the variable values, control flows, and method invocations. Such properties can be generalized to explain and diagnose software errors. This insight has been explored in designing the FailureDoc and FlowFixer techniques, as described in Chapter 3 and Chapter 6, respectively.

3. **Inferring high-level documentation to explain software failures.** To address the challenge of obtaining contextually-relevant information, our FailureDoc technique infers explanatory documentation as a debugging clue. Good documentation (i.e., code comments) can help developers quickly understand what source code does, facilitating program comprehension and software maintenance tasks [44]. When a test fails, the most useful documentation is relevant to the defect in the tested code and leads the developer to that defect. Later, when the test reveals a different defect, different documentation would be best. Such documentation informs developers of why the software fails, which kind of input leads the software to fail, and under which circumstances the software fails, rather than forcing developers to guess about what parts of the test and the tested code are relevant. Our FailureDoc technique utilizes this idea (Chapter 3). It leverages automated test generation and dynamic invariant detection techniques to generalize code observations into explanatory documentation.

4. **Approximating the intended software behavior by correct execution traces.** To address the challenge of specifying the intended behavior of a software system, our techniques use correct execution traces to approximate the intended software behavior. Correct execution traces contain valuable information about how a software system works or is intended to work, such as where the correct control flows take, and how the data values are related to one another. Thus, they can serve as an approximate testing oracle to check against the execution trace produced by the undesired program behavior. The differences between a correct execution trace (or a set of correct execution traces) and an undesired execution trace often reveal which part of the program goes wrong and why. This insight has been exploited in the ConfDiagnoser

and ConfSuggester configuration error diagnosis tools (Chapters 4 and 5). Both techniques correlate execution differences to the root-cause configuration options.

5. **Using user demonstration as a convenient way for human-tool interaction.** To address the challenge of communicating a software end-user's intention to a program analysis tool, our techniques employ user demonstration as a convenient way for human-tool interaction. Instead of requiring users to write testing oracles or code annotations in diagnosing an error, our error diagnosis techniques simply ask users to demonstrate the software usage scenarios to reproduce their encountered errors. Although less rigorous than a formal specification, demonstration is one of the simplest ways for end-users to naturally describe their intentions; it is easier than writing specifications or scripts of any form. User demonstration has been used in the ConfDiagnoser and ConfSuggester error diagnosis tools (Chapters 4 and 5) and the FlowFixer workflow repair tool (Chapter 6).

### *1.4 Scope*

The techniques and tools presented in this dissertation are for automated software testing and end-user error diagnosis. The activities of automated software testing are not limited to test generation and test failure explanation, which are our focus in this dissertation. Our testing techniques are designed for testing sequential programs executed on a single computer but not concurrent or distributed programs. Further, the testing techniques are specifically designed for testing an object-oriented program at the unit level and testing functional correctness but not other quality attributes such as performance and security. Our end-user error diagnosis approaches focus on errors that are deterministic and reproducible.

Chapter 8 discusses future directions of extending the techniques to analyze other types of software artifacts and to diagnose new types of software errors.

### *1.5 Outline*

The rest of this dissertation is organized as follows.

Chapter 2 describes the technique for automatically generating unit tests for software with constrained interfaces. The developed technique, called Palus, combines dynamic model inference,

static analysis, and random testing, to automatically generate tests that are legal and behaviorally-diverse. The empirical evaluation showed that tests generated by Palus achieved, on average, 16% higher coverage than existing techniques, and found more bugs. Palus was used internally at Google. It revealed 22 previously-unknown bugs from four large-scale, mature products.

Chapter 3 describes the technique for explaining test failures. The developed technique, called FailureDoc, assists developers in understanding a test failure by augmenting a failed test with contextually-relevant debugging clues: code comments that provide useful facts about the failure. In the experimental evaluation, FailureDoc generates meaningful documents for most failed tests. In a user study, FailureDoc reduced the time to understand a failed test by 14%, on average. A core developer of Apache Commons Math also confirmed its usefulness.

Chapter 4 describes the technique for automatically diagnosing software configuration errors. The developed technique, called ConfDiagnoser, uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options whose values should be corrected. The experiments showed that ConfDiagnoser achieved high diagnosis accuracy: for more than 70% of errors, the root cause was one of the top 3 suggestions; and more than half of the time, the root cause was the first suggestion. Further, ConfDiagnoser is able to diagnose non-crashing configuration errors that existing techniques failed to diagnose.

Chapter 5 describes the technique for diagnosing configuration errors for evolving software. The developed technique, called ConfSuggester, extends ConfDiagnoser to work on two different versions of the same software. ConfSuggester uses the desired behavior on the old software version as a baseline with which to compare new program behavior against, and only reasons about the behavioral differences. The evaluation showed that ConfSuggester is accurate in identifying the responsible configuration options to fix undesired behaviors on the new software version, and produced significantly better results than several existing or alternative techniques.

Chapter 6 describes the technique for automatically repairing broken workflows for GUI applications. FlowFixer uses dynamic profiling, static analysis, and random testing to suggest a replacement UI action that fixes a broken workflow. FlowFixer was applied to repair 16 real-world broken workflows from 5 mature GUI applications. It successfully repaired 15 workflows, while the best previous technique only repaired 6 workflows.

Chapter 7 surveys related work in three main categories, namely, program specification infer-

ence techniques, automated software testing techniques, and software error diagnosis and patching techniques.

Chapter 8 discusses several future research directions, including building user interfaces, conducting user studies, improving non-functional software properties, classifying and recovering from software errors, analyzing non-executable software artifacts, preventing end-user errors, and automating end-user programming. All of these topics will help program analysis techniques be further used for improving software reliability and error diagnosis in different scenarios.

Chapter 9 concludes with a summary of the contributions and a set of lessons learned.

# Chapter 2

# GENERATING TESTS FOR SOFTWARE WITH CONSTRAINED INTERFACES

This chapter describes a technique, called Palus, to automatically generate unit tests for software with constrained interfaces. We give an overview of the targeted problem in Section 2.1, show an illustrating example in Section 2.2, detail the Palus technique in Section 2.3, present the tool implementation in Section 2.4, describe the empirical evaluation in Section 2.5, and conclude in Section 2.6.

## *2.1 Problem*

In an object-oriented language like Java, a unit test consists of a sequence of method calls that explores a particular aspect of the behavior of the methods under test. Two primary objectives of testing are to achieve high structural coverage, which increases confidence in the code under test, and to find unknown bugs. To achieve either objective, unit testing requires desirable method-call sequences (in short, *sequences*) that create and mutate objects. These sequences generate target object states of the receiver or arguments of the method under test.

To reduce the burden of manually writing unit tests, many automated test generation techniques have been studied [10, 28, 31, 148, 191]. Of existing test generation techniques, bounded-exhaustive [22, 131], symbolic execution-based [65, 172, 201], and random [28, 148] approaches represent the state of the art. *Bounded-exhaustive* approaches generate sequences exhaustively up to a small bound on sequence length. However, generating good tests often requires longer sequences beyond the small bound. *Symbolic execution*-based analysis tools such as JPF [201] and CUTE [172] explore paths in the program under test symbolically and collect symbolic constraints at all branching points in an explored path. The collected constraints are solved if feasible, and a solution is used to generate an input for a specific method. However, these symbolic execution-based tools face the challenge of scalability, and the quality of generated inputs heavily depends on the test driver (which is often manually written [65, 172, 201]). *Random* approaches [28, 148]) are easy to use,

scalable, and able to find previously-unknown bugs [148], but they face challenges in achieving high structural coverage for certain programs. One major reason is that, for programs that have constrained interfaces, correct operations require calls to occur in a certain order with specific arguments. Thus, most randomly-created sequences may be illegal (violate the implicit specification) or may fail to reach new target states.

One possible way to improve the effectiveness of automated testing techniques is to use a formal specification to guide test generation [31, 68, 198]. However, such a specification is often absent in practice. Another way, which we follow, is to combine static and dynamic analysis. Static analysis examines program code and reasons over all possible behaviors that might arise at run time, while dynamic analysis operates by executing a program and observing the executions. Their mutual strengths can enhance one another [50].

Our Palus technique automatically generates *legal* and *behaviorally-diverse* tests. Palus operates in three steps: dynamic inference, static analysis, and guided random test generation. (1) In its dynamic component, Palus takes a correct execution as input, infers a call sequence model, and enhances it by annotating argument constraints. This enhanced call sequence model captures legal method-call orders and argument values observed from the sample execution. (2) In its static component, Palus computes dependence relations among the methods under test. The static analysis devised in Palus first identifies field access (read or write) information for each method, then uses the tf-idf weighting scheme [103] to weight the dependence between each pair of methods. In general, methods that read and write the same field are dependent, so testing them together has a higher chance of exploring different program behaviors. (3) Finally, both the dynamically-inferred model and the statically-identified dependence information guide a random test generator [148] to create legal and behaviorally-diverse tests.

Several past research tools follow an approach similar to ours, but omit one or two of the three stages of our approach. Randoop [148] is a pure random test generation tool. Palulu [10] is a representative of the dynamic-random approaches. It infers a call sequence model from a sample execution, and follows that model to create tests. However, the Palulu model lacks constraints for method arguments and has no static analysis phase to enrich the dynamically-inferred model with information about methods that are not covered by the sample execution. Finally, RecGen [243] uses a static-random approach. RecGen does not have a dynamic phase, and uses a static analysis to guide

random test generation. Lacking guidance from an accurate dynamic analysis, RecGen may fail to create legal sequences for programs with constrained interfaces, and may miss many target states (an example will be discussed in Section 2.2).

Unlike past research tools [10, 148, 243] mentioned above that only check generalized programming rules (e.g., the symmetry property of equality: `o.equals(o)`), Palus integrates with the JUnit theory framework [168]. This permits developers to write project-specific testing oracles for more effective bug finding.

### *2.2 Example*

We next give an overview of Palus with an illustrative example taken from the tinySQL [195] project, shown in Figure 2.1.

A tinySQL test must satisfy both *value* and *ordering* constraints. Creating a `tinySQLConnection` object requires a valid `url` string like `jdbc:tinySQL:fileNameOnDisk` to pass the check at line 4 or line 16[1]; and the `sql` string at line 23 must be a valid SQL statement like `select * from table_name`. Moreover, a legal sequence requires calling methods in a correct order: a valid `tinySQLDriver`, a `tinySQLConnection`, and a `tinySQLStatement` object must be created before a query can be issued. If a test generator fails to create arguments that satisfy these constraints, no `tinySQLConnection` and `tinySQLStatement` object will be created, no database query will be issued successfully, and the generated tests will fail to cover most of the code. As revealed by our experiment in Section 2.5, a pure random approach Randoop and a static-random approach RecGen could achieve only 29% line coverage for the tinySQL program.

A dynamic-random approach like Palulu [10] takes a (correct) sample execution as shown in Figure 2.2 as input, builds a call sequence model, and uses the model to guide a random test generator. The model inferred by Palulu from Figure 2.2 is shown in Figure 2.3. This model captures possible legal method-call orders (e.g., calling `executeQuery(String)` before `close()`) and argument values (e.g., the observed values of `url` and `sql` from the sample execution) to construct a sequence. After obtaining the model, Palulu uses a two-phase strategy to generate sequences. In the first phase, it creates sequences randomly *without* using the model, and keeps all generated sequences in a pool.

[1] Another constructor `tinySQLConnection()` creates an empty object with all fields uninitialized. To become a valid object, its fields need to be assigned separately.

```
1. public class tinySQLDriver implements java.sql.Driver {
2.   public tinySQLDriver() {}
3.   public Connection connect(String url, Properties info) {
4.     if(!acceptsURL(url) {return null;}
5.     return getConnection(info.getProperty("user"),url,this);
6.   }
     ...8 more methods omitted here...
   }
7. public class tinySQLConnection {
8.   protected String url, user, tsql;
9.   protected Driver driver;
10.  public tinySQLConnection() {}
11.  public tinySQLConnection(String user, String u, Driver d)
12.     throws SQLException {
13.     this.url = u;
14.     this.user = user;
15.     this.driver = d;
        //check the validity of url
16.     this.tsql = get_tinySQL();
     }
17.  public Statement createStatement() {
18.     return (Statement)new tinySQLStatement(this);
     }
     ... 20 more methods omitted here...
   }
19. public class tinySQLStatement {
20.   public tinySQLStatement(tinySQLConnection conn) {
21.     this.connection = con;
22.   }
23.   public ResultSet executeQuery(String sql) {
24.    //check the validity of this.connection
25.    //then issue the query and get the result
     }
     ... 20 more methods omitted here...
   }
```

Figure 2.1: Three classes taken from the tinySQL project [195]. Creating good sequences for this code requires invoking method calls in a specific order with specific arguments.

```
1. tinySQLDriver driver = new tinySQLDriver();
2. Connection conn = driver.connect("jdbc:tinysql:db", new Property());
3. tinySQLStatement stmt = new tinySQLStatement(conn);
4. stmt.executeQuery("select * from table_name");
5. stmt.close();
6. conn.close();
```

Figure 2.2: A sample method call sequence for the code in Figure 2.1. A dynamic-random approach can use this sample execution to build a legal call sequence model, as shown in Figure 2.3.

In the second phase, it creates sequences either by generating a new sequence from a model root (e.g., node A, B, or C in Figure 2.3) or extending an existing sequence along a model edge. If creating a sequence needs a type $T$ argument, it will *randomly* pick a sequence that creates a type $T$ value from the sequence pool.

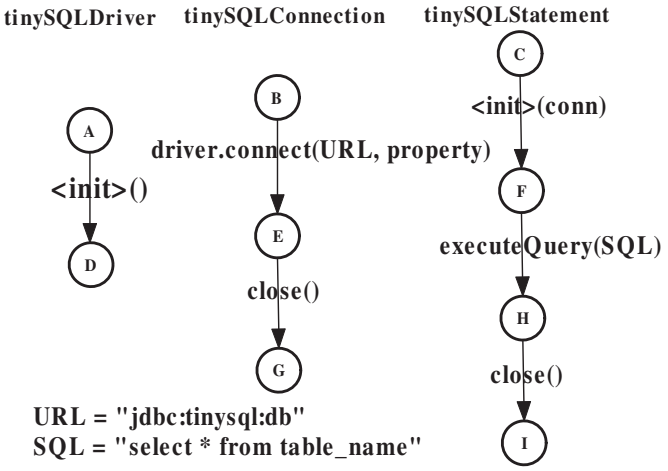



Figure 2.3: The call sequence model built by a dynamic-random approach (Palulu [10]) for classes in Figure 2.1 using the sample execution in Figure 2.2. Nodes A, B, and C represent the root node for each class. Potential nested calls are omitted here.

Although the Palulu approach offers a possible solution to create legal sequences, it faces several challenges to be effective in practice. First, the random phase in Palulu may end up with few legal sequences. For example, it is very likely to generate no `tinySQLStatement` and `tinySQLConnection` objects for the motivating example. Second, the call sequence model in Figure 2.3 does not consider constraints for non-primitive arguments. For example, the constructor of `tinySQLStatement` needs to have a `tinySQLConnection` object with all its fields correctly initialized as an argument. However, the Palulu-inferred model does not keep such constraints. Therefore, Palulu will pick an existing `tinySQLConnection` object randomly from the pool, and thus may fail to create a valid `tinySQLStatement` object. Third, the sequences it generates are restricted by the dynamically-inferred model, which tends to be incomplete. For instance, a correct sample execution would never contain an exception-throwing trace like:

```
tinySQLStatement.close();

tinySQLStatement.executeQuery("select * from ..");
```

Thus, Palulu may fail to generate sequences that either have different method-call orders from the inferred model or contain methods not covered by the sample execution. In our experiment as described in Section 2.5, although Palulu outperforms Randoop and RecGen, it achieved only 41%

code coverage for the tinySQL program.

Our Palus technique remedies the above problems by enhancing the call sequence model with argument constraints and performing a static analysis to find dependent methods that should be tested together (including those not covered by the model).

Take the code snippet in Figure 2.1 as an example. Palus first constructs a call sequence model (Figure 2.6) and annotates it with two kinds of constraints, namely, *direct state transition dependence constraints* and *abstract object profile constraints* (Section 2.3.1). These two constraints permit Palus to find a desirable object for creating a legal sequence as well as a group of distinct objects. Then, Palus performs a static analysis to identify method dependence relations based on the fields they may read or write, and prefers related methods when extending a model sequence. Testing methods that access the same field together may have a higher chance to explore different program states. For example, let us assume method `executeQuery` in class `tinySQLStatement` reads field `connection`, and method `close` writes field `connection` (assigns it to `null`). Thus, when testing method `executeQuery`, Palus may recommend to invoke method `close` before it, since testing them together permits reaching a new program state, and checks the behavior of executing a query after closing a connection. This combined static and dynamic test generation strategy permits Palus to generate more effective tests. For the tinySQL program, Palus achieves 59% code coverage, which is 30% higher than Randoop and RecGen and 18% higher than Palulu.

Palus also integrates with the JUnit theory framework [168], which permits developers to clarify design intentions and write project-specific testing oracles such as the one in Figure 2.4. Palus will automatically execute this theory and check its output when finding a `tinySQLStatement` object and a `String` object.

## *2.3 Technique*

Palus consists of four components (Figure 2.5), namely, a load-time instrumentation component and a dynamic model inference component (dynamic analysis, Section 2.3.1), a static method analysis component (static analysis, section 2.3.2), and a guided random test generation component (Section 2.3.3).

The load-time instrumentation instruments the Java bytecode. The dynamic model inference

```
@Theory
void noActionAfterClose(tinySQLStatement stmt,String sql){
    Assume.assumeNotNull(stmt);
    Assume.assumeNotNull(sql);
    stmt.close();
    try {
        stmt.executeQuery(sql);
        fail("The statement should have been closed!");
    } catch (Exception e) {
        //ok here
    }
}
```

Figure 2.4: A testing oracle expressed as a JUnit theory. For any non-null `tinySQLStatement` object, the `close` method should not throw any exception, after which `executeQuery` must throw an exception.

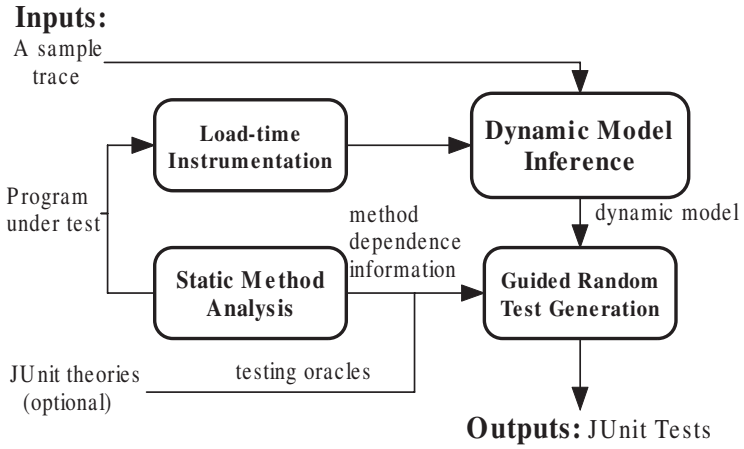



Figure 2.5: Architecture of the Palus tool.

component collects traces from the sample execution and builds an enhanced call sequence model for each class under test. The static method analysis component computes the set of fields that may be read or written by each method, and then calculates pairwise method dependence relevance values. Finally, the dynamically-inferred model and method dependence information together guide the random sequence generation process.

### *2.3.1 Dynamic Analysis: Model Inference from Sample Executions*

We first briefly introduce the call sequence model, then present our enhancement.

*Call Sequence Model Inference*

A call sequence model [10] is a rooted, directed, acyclic graph. Each model is constructed for one class observed during execution. Edges (or *transitions*) represent method calls and their arguments[2], and each node in the graph represents a collection of object states, each of which may be obtained by executing the method calls along some path from the root node. Each path starting at the root corresponds to a sequence of calls that operate on a specific object — the first method constructs the object, while the rest mutate the object (possibly as one of their parameters).

The construction of the call sequence model is object-sensitive. That is, the construction algorithm first constructs a call sequence graph for each object of the class observed during execution. Then, it merges all call sequence graphs of objects of the class. Thus, the final call sequence model is a summary of the call sequence graphs for all instances of the class. The call sequence model handles many Java features like nested calls, recursion, and private calls. The detailed definition, construction steps, and inference algorithms appear in [10].

*Model Enhancement with Argument Constraints*

The call sequence model is too permissive: using it can lead to creating many sequences that are all illegal and thus have similar, uninteresting behavior. Our approach enhances a call sequence model with two kinds of method argument constraints. *Direct state transition dependence constraints* are related to how a method argument was created. *Abstract object profile constraints* are related to the value of a method argument's fields.

***Direct State Transition Dependence Constraint.*** This constraint represents the state of a possible argument. An edge from node A to B indicates that an object state at A may be used as an argument when extending a model sequence at B.

Consider the sample execution in Figure 2.2. The `Driver` object used at line 2 is the result produced by line 1, and the `Connection` object used at line 3 is the execution outcome of line 2. Therefore, our approach adds two *direct state transition dependence* edges to the original call sequence model, and results in an enhanced model shown in Figure 2.6. Using the enhanced model, when a test

[2]The original model only keeps primitive and string arguments, and we will extend it to other argument types in the next section.

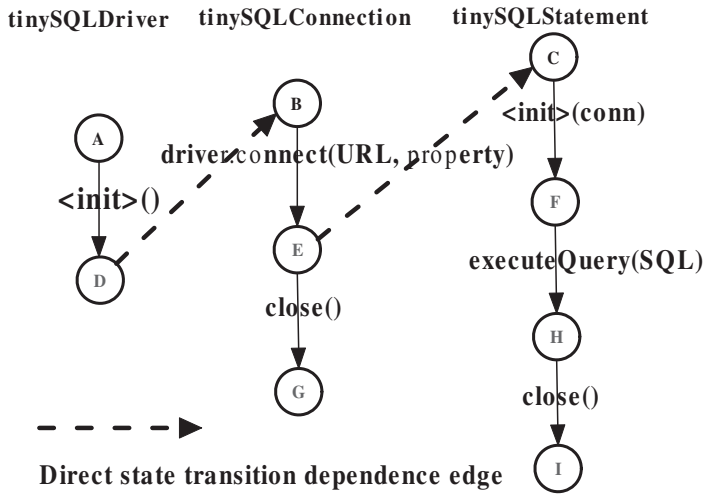



Figure 2.6: An enhanced call sequence model with direct state transition dependence edges for the example in Figure 2.1.

generation tool is creating a sequence (for example, calling method `tinySQLStatement(conn)`), it will follow the dependence edge backward to an argument.

A state dependence edge may be too strict: it indicates an exact sequence of method calls that must be used to construct an object. However, a different object whose value is similar, but which was constructed in a different way, may also be legal and permit the method call to complete non-erroneously. To address this problem, we use a lightweight abstract object profile representation as another argument constraint.

***Abstract Object Profile Constraint.*** For an object, we define its *concrete state* as a vector, $v = \langle v_1, ..., v_n \rangle$, where each $v_i$ is the value of an object field. An abstract object profile maps each field's value to an abstract value. Formally, we use a state abstraction function which maps concrete value $v_i$ to an abstract value $s_i$ as follows:

- Concrete numerical value $v_i$ (of type `int`, `float`, etc.), is mapped to three abstract values $v_i < 0$, $v_i = 0$, and $v_i > 0$.
- Array value $v_i$ is mapped to four abstract values $v_i =$ `null`, $v_i$ is empty, $v_i$ contains `null`, and all others.
- Object reference value $v_i$ is mapped to two abstract values $v_i =$ `null`, and $v_i \neq$ `null`.

- Boolean and enumeration values are not abstracted. In other words, the concrete value is re-used as the abstract value.

For example, let us assume a `tinySQLConnection` object has four fields `url`, `user`, `tsql`, `driver` as in Figure 2.1. A valid `tinySQLConnection` object might thus have a state $v$ = ⟨`'jdbc:tinysql:file'`, `'user'`, `'select * from table'`, `new tinySQLDriver()`⟩, which corresponds to an abstract state $s$ = ⟨≠`null`, ≠`null`, ≠`null`, ≠`null`⟩.

When our tool Palus builds a call sequence model from a sample execution, it computes the abstract object profiles for all arguments and keeps them in the model. If Palus is unable to obtain a value created by a given sequence of method calls, then it instead uses one that matches the abstract state profile.

Even coarse profiles help prevent the selection of undesirable objects as arguments. For example, if we run Palulu on the tinySQL subject programs for 20 seconds, it creates 65 `tinySQLConnection` objects. However, only 5 objects have legal state, that is, a state corresponding to the observed abstract state ⟨≠`null`, ≠`null`, ≠`null`, ≠`null`⟩. The remaining 60 objects have the abstract state ⟨=`null`, =`null`, =`null`, =`null`⟩. A tool like Palulu that randomly picks a `tinySQLConnection` object is very likely to end with an illegal object instance. In contrast, Palus has a higher chance of getting a legal `tinySQLConnection` object, because it selects a `tinySQLConnection` object with an abstract state ⟨≠`null`, ≠`null`, ≠`null`, ≠`null`⟩.

### *2.3.2 Static Analysis: Model Expansion with Dependence Analysis*

The dynamically-inferred model provides a good reference in creating legal sequences. However, the model is only as complete as the observed executions and may fail to cover some methods or method-call invocation orders. To alleviate this limitation, we designed a lightweight static analysis to enrich the model. Our static analysis hinges on the hypothesis that two methods are related if the fields they read or write overlap. Testing two related methods has a higher chance of exploring new program behaviors and states. Thus, when extending a model sequence, our approach prefers to invoke related methods together.

*Method Dependence Analysis*

Our approach computes two types of dependence relations: *write-read* and *read-read*.

***Write-read*** *relation*: If method `f` reads field `x` and method `g` writes field `x`, we say method `f` has a *write-read* dependence relation on `g`.

***Read-read*** *relation:* If two methods `f` and `g` both *read* the same field `x`, each has a *read-read* dependence relation on the other. Two methods that have a *write-read* dependence relation may also have a *read-read* dependence relation.

In our approach, we first compute the read/write field set for each method, then use the following strategy to merge the effects of the method calls: if a callee is a private method or constructor, we recursively merge its access field set into its callers. Otherwise, we stop merging. This strategy is inspired by the common coding practice that public methods are a natural boundary when developers are designing and implementing features [52, 133].

*Method Relevance Ranking*

One method may depend on multiple other methods. We define a measurement called *method dependence relevance* to indicate how tightly each pair of methods is coupled. In the method sequence generation phase (Section 2.3.3), when a method `f` is tested, its most dependent methods are most likely to be invoked after it.

We used the tf-idf (term frequency – inverse document frequency) weighting scheme [103] to measure the importance of fields to methods. In information retrieval, the tf-idf weight of a word $w$ in a document $doc$ statistically measures the importance of $w$ to $doc$. The importance increases proportionally to the frequency of $w$'s appearance in $doc$, but decreases proportionally to the frequency of $w$'s appearance in the corpus (all documents). Our approach treats each method as a document and each field read or written as a word.

The *dependence relevance* $W(m_k, m_j)$ between methods $m_k$ and $m_j$ is the sum of the tf-idf weights of all fields, $V_{m_k \to m_j}$, via which $m_k$ depends on $m_j$.

$$W(m_k, m_j) = \sum_{v_i \in V_{m_k \to m_j}} tfidf(v_i, m_k)$$

**Auxiliary Methods:**
CoveredMethods($models$): return all model-covered methods
DivideTime($timelimit$): divide timelimit into 2 fractions
RandomMember($set$): return a set member randomly
RandomGenerate($method$): create a sequence for method using pure random test generation [148]
**GenSequences**($timelimit$, $methods$, $models$, $dependences$)

1: $uncovered\_methods \leftarrow methods \setminus$ CoveredMethods($models$)
2: $reachingSeqs \leftarrow$ new Map⟨ModelNode, List⟨Sequence⟩⟩
3: $t_{model\text{-}guided}$, $t_{unguided} \leftarrow$ DivideTime($timelimit$)
4: seqs $\leftarrow \{\}$
5: **while** $t_{model\text{-}guided}$ not expired **do**
6: Randomly perform one of the following two actions:
7: 1. $root \leftarrow$ RandomMember($models$)
8: $tran \leftarrow$ RandomMember($root.transitions$)
9: $newSeq \leftarrow$ GuidedSeqGen($tran$, $dependences$)
10: $seqs \leftarrow seqs \cup newSeq$
11: $reachingSeqs$[$trans.dest\_node$].add($newSeq$)
12: 2. $node \leftarrow$ RandomMember($reachingSeqs$.keySet())
13: $seq \leftarrow$ RandomMember($reachingSeqs$[$node$])
14: $tran \leftarrow$ RandomMember($node.transitions$)
15: $newSeq \leftarrow$ GuidedSeqGen($tran$, $dependences$)
16: $seqs \leftarrow seqs \cup newSeq$
17: $reachingSeqs$[$node$].remove($seq$);
18: $reachingSeqs$[$trans.dest\_node$].add($newSeq$);
19: **end while**
20: **while** $t_{unguided}$ not expired **do**
21: $method \leftarrow$ RandomMember($uncovered\_methods$)
22: $seqs \leftarrow seqs \cup$ RandomGenerate($method$)
23: **end while**
24: **return** $seqs$

Figure 2.7: Sequence generation algorithm in Palus. The procedure GuidedSeqGen is shown in Figure 2.8.

The intuition is that the dependence relevance between two methods is determined by the fields they both access, as well as these fields' importance to the dependent method.

### *2.3.3 Guided Test Generation*

*Sequence Generation*

The sequence generation algorithm listed in Figure 2.7 takes four arguments, namely *a time limit*,

**Auxiliary Methods:**
GetDependentMethod($m$): return a method, randomly chosen with probability proportional to its dependence relevance with $m$
CreateSequenceForConst($value$): create a new sequence that produces the given value
GetSequenceByState($dep\_state$): get an existing sequence that produces a result of the given $dep\_state$
GetSequenceByProfile($profile$): get an existing sequence that produces a result of the given profile
GetSequenceByType($T$): get an existing sequence that produces a result of type $T$
Concatenate($param\_seqs$, $m_1$, $m_2$): concatenate $param\_seqs$, $m_1$, $m_2$ to form a new sequence
**GuidedSeqGen**($transition$, $dependences$)

```
1:  m(T_1, ... T_n) ← transition.method
2:  param_seqs ← new List⟨Sequence⟩
3:  for each T_i in (T_1, ..., T_n) do
4:     if T_i is primitive or string type then
5:        seq ← CreateSequenceForConst(transition.recordedValue)
6:     else
7:        ⟨dep_state, profile⟩ ← transition.constraints
8:        seq ← GetSequenceByState(dep_state)
9:        if seq = null then
10:          seq ← GetSequenceByProfile(profile)
11:       end if
12:       if seq = null then
13:          seq ← GetSequenceByType(T_i)
14:       end if
15:    end if
16:    param_seqs ← param_seqs ∪ seq
17: end for
18: dep_m(T'_1, ..., T'_n) ← GetDependentMethod(m)
19: for each T'_i in (T'_1, ..., T'_n) do
20:    param_seqs ← param_seqs ∪ GetSequenceByType(T'_i)
21: end for
22: return Concatenate(param_seqs, m, dep_m);
```

Figure 2.8: Guided sequence generation algorithm in Palus.

*a set of methods* for which to generate sequences, *enhanced call sequence models* constructed from observed sample executions (Section 2.3.1), and *method dependence relations* (Section 2.3.2).

The algorithm shown in Figure 2.7 works in two phases: the first phase for methods that were executed in the sample execution and so appear in the model, and the second phase for other methods. The generator creates sequences incrementally, maintaining a set of generated sequences (lines 4,

10, 16, and 22). The algorithm also maintains a node-sequence mapping that maps each node in a call sequence model to a list of sequences that end at that node (lines 2, 11, 17, and 18). When generating a sequence, the algorithm either selects an edge that is going out of a model root and creates a sequence for it (lines 7–11), or extends an existing sequence (lines 12–18). After extending an existing sequence (line 15), the existing sequence is removed from the node-sequence map (line 17) and the newly-created sequence is added to the map (line 18). The remove prevents Palus from repeatedly extending a single sequence. The extended sequence actually has had two calls, not one, added to it. It is added to the map as if the second call did not affect its state in the call sequence model; if the assumption is not true, then this addition can add further variability to the generated tests. Finally, to avoid missing model-uncovered methods, our algorithm randomly creates sequences for model-uncovered methods (lines 20–23). In our experiments, $t_{model\text{-}guided} : t_{unguided} = 4 : 6$.

The algorithm in Figure 2.8 shows steps in creating a sequence using both dynamic model and static dependence information. Given a tested method $m$ (line 1), the algorithm uses constraints (recorded in the dynamic model) to find sequences for its parameters (lines 5, 8, 10, and 13). Then, the algorithm queries the dependence information to get a dependent method $dep\_m$ and find parameter sequences for it (lines 19–21). Finally, the parameter sequences, tested method, and its dependent method are concatenated to form a new sequence (lines 22).

Every sequence returned by `GenSequences` is executed and checked against a set of generalized properties [148] as well as user-provided oracles (Section 2.3.3). Any violating tests are classified as failure-revealing tests.

*Oracle Checking*

Palus integrates the JUnit theory framework [168] (which is similar to Parameterized Unit Tests [192]), permitting developers to write domain-specific oracles. A theory is a generalized assertion that should be true for any data.

Take the theory in Figure 2.4 as an example, Palus will automatically check for every non-null `tinySQLStatement` object and non-null `String` value pair, that the assertion should hold. When Palus executes a theory with concrete object values, if an `Assume.assume*` call fails and throws an `AssumptionViolatedException`, Palus intercepts this exception and silently proceeds. If some

generated inputs cause an `Assert.assert*` to fail, or an exception to be thrown, the failures are outputted to the developer.

*Annotation Support for Additional Inputs*

Palus captures the primitive and string type argument values from the observed sample execution in the same way as Palulu does. However, a test generator may still need more additional inputs to explore certain behaviors of the program under test. For example, the sample execution in Figure 2.2 only covers a special case of *selecting all tuples in the example table* in testing method `executeQuery(String)`. It would be useful if a test generator could feed more different input values (a valid string) to the tested method. Palus provides an annotation called `@ParamValue` to permit developers to specify special values for a specific tested method, and will use such user-provided values to create new sequences. For example, a developer could write an annotation like:

```
@ParamValue(cn="tinySQLStatement",mn="executeQuery")
public static String queryMore = "select * from table_name where id > 10";
```

When Palus creates a sequence for the `executeQuery` method in class `tinySQLStatement`, it will also use the value of `queryMore`.

## *2.4 Tool Implementation*

We implemented Palus using the ASM bytecode analysis framework and the Randoop test generation tool [148]. Palus uses ASM to perform load-time instrumentation of Java bytecode, to record the execution trace. The method dependence analysis is also performed at the bytecode level. Palus works in a fully automatic and push-button way to generate unit tests, and scales to realistic programs. The test generation phase requires from the user a time limit, a sample execution trace, and a list of Java class names. Optionally, the user may specify additional theories for oracle checking (Section 2.3.3), and `@ParamValue` annotations (Section 2.3.3).

To increase robustness, Palus spawns a test generation process and monitors its execution. If it hangs before the user-specified time, Palus spawns a new process, using a new random seed. Method sequences generated before Palus crashes are also output to users.

## *2.5 Evaluation*

We compared tests generated by Palus with tests generated by Randoop (a pure random approach), Palulu (a dynamic-random approach), and RecGen (a static-random approach). We compare the tests' structural coverage and bug detection ability. Finally, we discuss the suitability of these four tools to programs with different characteristics.

We also report on experience using Palus at Google (Section 2.5.5). Palus found 22 unique previously-known bugs in four well-tested projects from Google's codebase, including the popular Google Buzz, AdWords, and Orkut products. This experience demonstrates the applicability of Palus to real-world, industrial-strength software products.

### *2.5.1 Research Questions*

We investigated the following two research questions:

**RQ1: Test coverage.** Do tests generated by Palus achieve higher structural coverage than an existing pure random test generation tool (Randoop), dynamic-random test generation tool (Palulu), and static-random test generation tool (RecGen)? This research question helps to demonstrate how dynamic and static analyses help in improving automated test generation.

**RQ2: Bug finding.** Do tests generated by Palus detect more bugs than tests generated by Randoop, Palulu, and RecGen? This research question helps to demonstrate the bug finding ability of our approach.

### *2.5.2 Subject Programs and Tools*

We evaluated Palus on six open-source Java programs (Table 2.1). tinySQL [195] is a lightweight SQL engine implementation that includes a JDBC driver. SAT4J [170] is an efficient SAT solver. JSAP [105] is a simple argument parser, which syntactically validates a program's command line arguments and converts those arguments into objects. Rhino [163] is an implementation of JavaScript, which is typically embedded into Java applications to provide scripting to end-users. BCEL [17] is a Java bytecode manipulation library that lets users analyze and modify Java class files. Apache Commons [4] extends the JDK with new interfaces, implementations, and utilities.

| Program (version) | Lines of code | Classes | Methods |
|---|---|---|---|
| tinySQL (2.26) | 7672 | 31 | 702 |
| SAT4J (2.2.0) | 9565 | 120 | 1320 |
| JSAP (2.1) | 4890 | 91 | 532 |
| Rhino (1.7) | 43584 | 113 | 2133 |
| BCEL (5.2) | 24465 | 302 | 2647 |
| Apache Commons (3.2) | 55400 | 445 | 5350 |

Table 2.1: Subject Programs used in evaluating Palus.

The top five subject programs in Table 2.1 contain many constraints for writing legal tests, while the last subject program (Apache Commons) is designed as a general utility library for developers, and thus has few constraints on its methods. We compare the effectiveness of Palus with three other closely related tools on the above set of subject programs, and investigate the suitability of these four tools on programs with different characteristics.

Randoop implements a technique called feedback-directed random testing [148]. Its key idea is to execute each test as soon as it is generated, and use information obtained from execution to guide the test generation process as it creates further tests. Illegal sequences like `sqrt(-1)` in which the argument is required to be non-negative will be discarded immediately.

Palulu [10] is a typical dynamic-random approach. Since the existing Palulu implementation relies on a deprecated version of Randoop, we re-implemented its algorithm in Palus[3].

RecGen [243] is a representative of static-random approaches. It uses the result of a static analysis to guide random test generation by recommending closely related methods.

### *2.5.3 Experimental Procedure*

To answer **RQ1**, we ran Palus, Randoop, Palulu, and RecGen on each subject with a time limit of 100 seconds per class (not per subject program). Increasing the time limit does not noticeably increase coverage for any of the tools. We then ran the generated tests and used Cobertura [35] to measure line/branch coverage.

[3]Our re-implementation of Palulu achieves higher coverage than the previous one: for example, 41% and 44% line coverage for tinySQL and SAT4J, respectively, compared to 32% and 36% for the previous implementation [10].

| Program | Steps to obtain a sample trace | Coverage (%) line | branch |
|---|---|---|---|
| tinySQL | Execute 10 simple SQL statements | 40 | 30 |
| SAT4J | Solve a satisfiable formula of 5 clauses | 20 | 12 |
| JSAP | Run its unit test suite | 35 | 30 |
| Rhino | Interpret a stream of JavaScript code | 19 | 10 |
| BCEL | Parse one class | 17 | 8 |
| Apache Commons | Run a subset (484 tests) of its unit test suite | 20 | 16 |

Table 2.2: Steps to obtain a sample execution, and its coverage.

Palus and Palulu require a sample trace to infer a model. We ran each subject's existing unit test suites, or, if no test suite is available, one simple example from their user manuals. Palus and Palulu use the same execution trace for test generation. Table 2.2 summarizes the steps to obtain a sample trace, and its coverage.

For **RQ2**, we conducted two separate experiments. First, we executed the JUnit tests generated by the four tools and measured the number of unique bugs they find. Second, we wrote several simple specifications using the JUnit theory framework, and ran Palus on each subject again. We manually checked each failed test to verify those that revealed bugs. Finally, we compared the bug-finding ability of the four tools on four well-tested Google products.

### *2.5.4 Coverage Results*

Figure 2.9 compares Randoop, Palulu, RecGen, and Palus on the subject programs.

*Comparison with a Pure Random Approach*

Palus outperforms Randoop on 5 out of 6 subjects in both line and branch coverage, and achieves the same coverage for the last subject (Apache Commons). There are two primary reasons for this improvement.

First, guided by the inferred model, it is easier for Palus to construct legal sequences. Sequences like creating a database connection in the tinySQL subject, initializing a solver correctly in the SAT4J subject, and generating a syntactically-correct JavaScript program in the Rhino subject all require invoking method calls in a specific order with specific arguments. Such sequences are difficult to

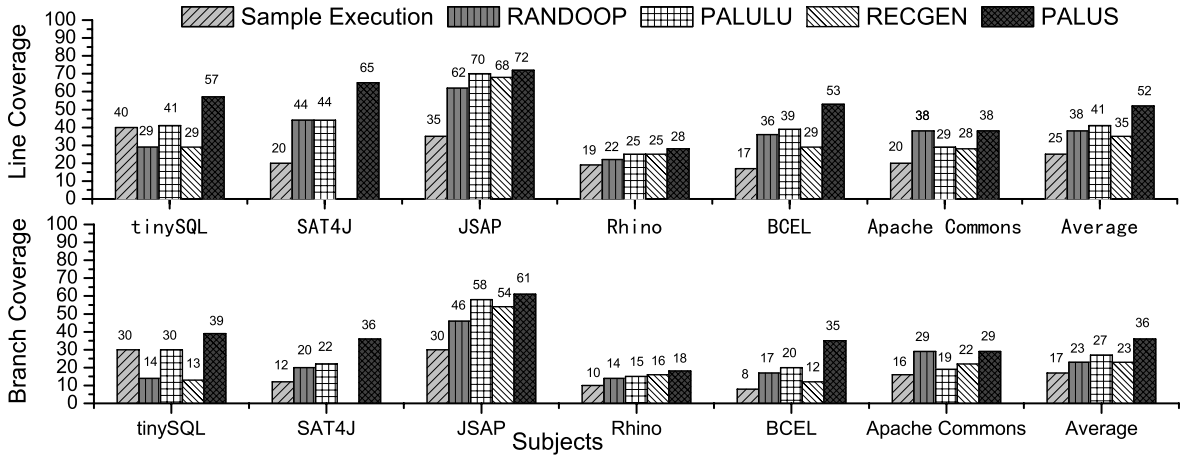



Figure 2.9: Structural coverage of tests generated by Randoop, Palulu, RecGen, and Palus. For each subject, coverage of the sample execution is also shown. RecGen fails to generate compilable tests for SAT4J. The average coverage results are shown at the far right.

create by the randomized algorithm implemented in Randoop. In fact, most of the tests generated by Randoop for the first five subjects in Table 2.1 only cover error-handling statements/branches.

Second, the static analysis used in Palus helps to diversify sequences on a specific legal path by testing related methods together. This helps Palus to reach more program states. For the last subject program, Apache Commons, Palus actually *falls back* to Randoop. The reason is that, Apache Commons is designed as a general library and has very few constraints that developers must satisfy. For example, one can put any object into a container without any specific method invocation order or argument requirement. Therefore, the information provided by the sample execution trace is not very useful for Palus. Randomly invoking methods and finding type-compatible argument values across the whole candidate domain could achieve the same results.

*Comparison with a Dynamic-Random Approach*

Tests generated by Palus achieve higher line/branch coverage than Palulu for all six subjects. For most subjects, Palulu creates a legal sequence with the assistance of the inferred model. However, there are three major reasons for the result difference. First, the pure random generation phase in Palulu creates many illegal object instances, which pollute the sequence pool. Second, the model inferred by Palulu lacks necessary constraints for method arguments, so that it is likely to pick up

invalid objects from the potentially polluted sequence pool and then fail to reach desirable states. Third, Palulu does not use static analysis to diversify a legal sequence by appending related methods, and may miss some target states.

For example, in SAT4J, Palus achieves 81% line coverage and 43% branch coverage for the `org.sat4j.minisat.constraints.cnf` package, while Palulu only achieves 39% and 21%, respectively. The reason is that class constructors in this package often need specific argument values. For instance, the constructor of class `BinaryClause` only accepts an `IVetInt` (an integer vector wrapper) object with exactly 2 elements. Thus, without explicit argument constraints, choosing an arbitrary `IVetInt` value could easily fail the object construction. We also find the static analysis phase to be useful. For example, SAT4J's `DimacsSolver.addClause(IVecInt literals)` method first checks a boolean flag `firstConstr`, then takes different branches according to its value. When testing this method, Palus identifies another method `reset()`, which sets the `firstConstr` be `false`, as dependent. Palus invokes `reset()` before calling `addClause(IVecInt literals)` in some tests to make the generated sequences reach more states.

Palus achieves little coverage increase over Palulu in two subjects, namely JSAP and Rhino. We identified two possible reasons. First, JSAP does not have as many constraints as other subjects in creating legal sequences. The only constraints are to check the validity of user inputs from command line before parsing (e.g., users need to provide a well-formatted string input). Moreover, most methods in JSAP perform string manipulation operations, and do not have much dependence between each other. Thus, Palus achieves the same coverage in most of the non-constrained code. Second, over 90% of the source code in Rhino belongs to the JavaScript parser and code optimizer modules. Code in these two modules will not be executed unless the interpreter is provided with a program with certain code structure. Only specific code structures trigger the optimizer to perform certain code transformations. However, the example program we used in the experiment to obtain an execution is a simple JavaScript program without any sophisticated structures. Therefore, the model Palus inferred failed to guide the test generator to cover most of the sophisticated code transformation and optimization part.

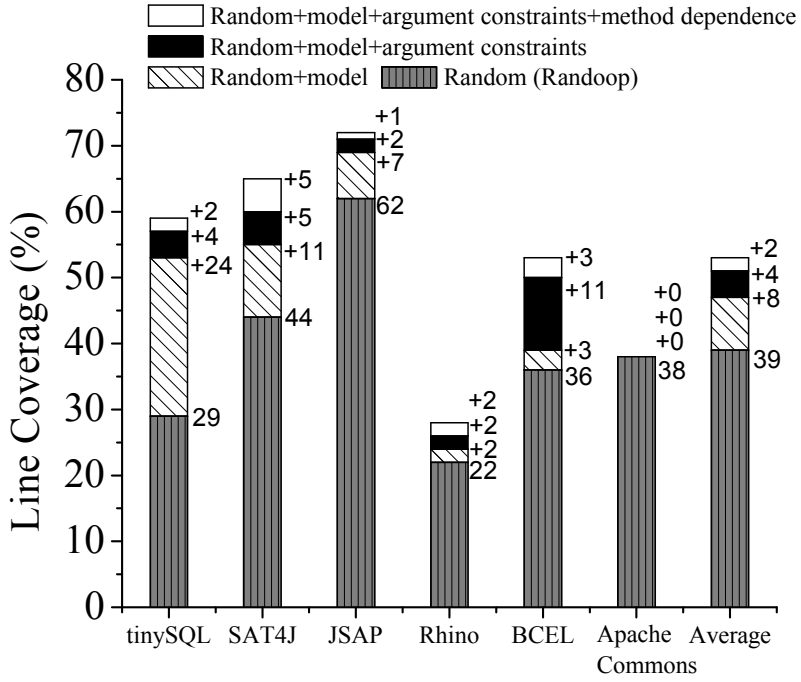



Figure 2.10: Breakdown of the line coverage increase for 6 subjects in table 2.1. The average result is shown at the far right. Branch coverage shows a similar pattern, which we omit for brevity.

*Comparison with a Static-Random Approach*

Tests generated by Palus achieve higher coverage than RecGen for all subjects. For subjects tinySQL and BCEL, Palus almost doubles the coverage. The primary reason is that, although RecGen uses a well-designed static analysis to recommend related methods to test together, it fails to construct legal sequences for most of the subjects with constrained interfaces.

Another problem is that the latest release of RecGen is built on top of a deprecated version of Randoop that contains bugs that have since been fixed. Thus, RecGen fails to generate compilable tests for the SAT4J subject.

*Coverage Increase Breakdown*

Figure 2.10 shows the line coverage increase contributed by each component in Palus. Using a call sequence model in Palus's algorithm (Figure 2.8) improves the line coverage percentage

by 0–24%, compared to a random approach. Enhancing the call sequence model with argument constraints improves line coverage percentage by another 0–11%. The static method dependence analysis further improves line coverage percentage by 0–5%. Note that in Figure 2.10, Random + model is better than Palulu [10], since the pure random generation phase in Palulu pollutes the sequence pool.

Overall, by combining dynamic inference and static analysis, Palus improves the line coverage on average by 14% (min = 0% for Apache Commons, max = 30% for tinySQL) over a pure random approach.

*Input Execution Trace and Test Coverage*

We next investigated how sensitive are the test coverage results of Palus to its input execution traces. We chose tinySQL and SAT4J as two examples, since the input trace size can be easily controlled. For tinySQL, we executed all the available SQL statements (23 in total) from its user manual to obtain a more comprehensive trace, and then used that to guide Palus. The generated tests achieved slightly higher structural coverage (60% line coverage and 41% branch coverage) than the results reported in Figure 2.9 (59% line coverage and 40% branch coverage). For SAT4J, we obtained two additional execution traces by solving two complex formulas from its example repository [170], containing 188 clauses and 800 clauses, respectively. Using the formula with 188 clauses, Palus achieved 65% line coverage and 37% branch coverage. Using the formula with 800 clauses, Palus achieved 66% line coverage and 37% branch coverage. Compared to the results in Figure 2.9 by using a formula with 5 clauses (65% line coverage and 36% branch coverage), the coverage increase was slight. Although we have not tried all available examples in SAT4J's code repository, which contains millions of clauses in total, these two experiments indicate that a simple trace is still useful for Palus, and suggest that Palus's results is not very sensitive to its input execution trace.

*Model Size and Test Coverage*

We further investigated the trade-off between model size and generated test coverage, by applying the *kTail* algorithm [21] to refine Palus's model. The *kTail* algorithm merges model nodes whose equivalence is computed by comparing their tails of length $k$, where the tails of length $k$ are the

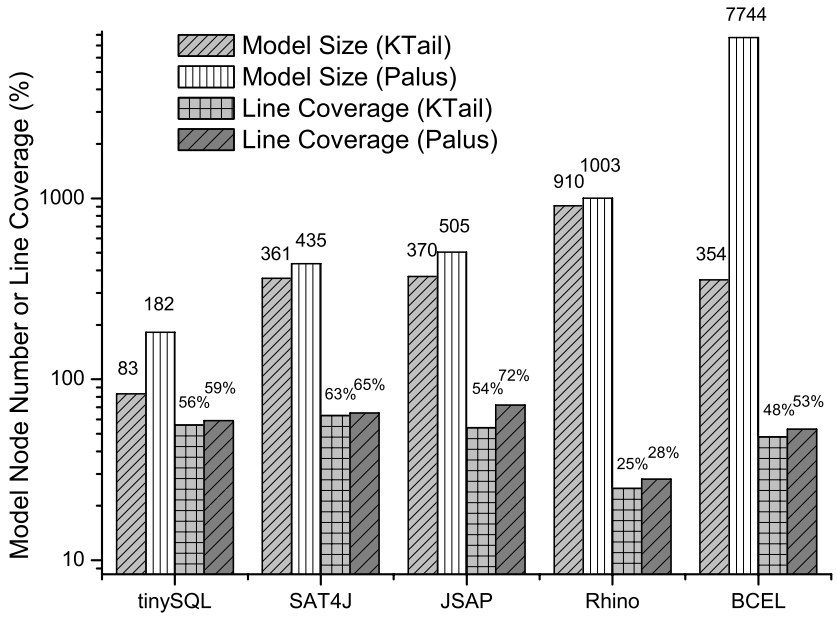



Figure 2.11: Model size (node number in log scale) and test coverage after applying the *kTail* algorithm with $k$ = 3. The line coverage data for Palus is taken from Figure 2.9.

method invocation sequences that exit the model node and include at most $k$ method invocations.

We chose $k$ = 3 in our experiment to generalize the inferred model, since a value between 2 and 4 is recognized as a reasonable choice in the absence of additional information [129]. We then used the refined model to guide Palus. Figure 2.11 shows the experimental results. The *kTail* algorithm can substantially reduce the model size by 9%–95%, but also decreases the test coverage by 2%–18%.

The purpose of a dynamically-inferred model in Palus is to guide a test generator to create legal method-call sequences, not for human inspection to understand program behavior like [41, 129]. Thus, the model size is not crucial to Palus.

### *2.5.5 Bug Finding Ability*

We compared the bug finding ability of each tool on both six open-source subjects (Table 2.1) and four well-tested Google products (Table 2.3). Table 2.4 gives the experimental results.

| Subject | Classes |
|---|---|
| Google Testing Server | 238 |
| Google Buzz | 600 |
| Google AdWords | 1269 |
| Google Orkut | 1455 |

Table 2.3: Four Google products used to evaluate Palus. Column "Classes" is the number of classes we tested, which is a subset of each product.

| Subject Programs | Number of bugs found | | | |
|---|---|---|---|---|
| | Randoop | Palulu | RecGen | Palus |
| tinySQL | 1 | 1 | 1 | 1 |
| SAT4J | 1 | 1 | – | 1 |
| JSAP | 0 | 0 | 0 | 0 |
| Rhino | 1 | 1 | 1 | 1 |
| BCEL | 74 | 70 | 37 | 74 |
| Apache Commons | 3 | 3 | 3 | 3 (4*) |
| *Total* | 80 | 76 | 42 | 80 (81*) |
| | | | | |
| Google Products | Number of bugs found | | | |
| | Randoop | Palulu | RecGen | Palus |
| Testing Server | 2 | 2 | – | 4 |
| Buzz | 2 | 2 | – | 3 |
| AdWords | 2 | 2 | – | 2 |
| Orkut | 13 | 12 | – | 13 |
| *Total* | 19 | 18 | – | 22 |

Table 2.4: Bugs found by Randoop, Palulu, RecGen, and Palus in the programs of Tables 2.1 and 2.3. The starred values include an additional bug Palus found with an additional testing oracle written as a JUnit theory. RecGen fails to generate compilable tests for the SAT4J subject, and can not be configured to work properly in Google's testing environment.

*Bugs in Experimental Subjects*

The first part of this evaluation compares the number of unique bugs found by Randoop, Palulu, RecGen, and Palus using default contracts as testing oracles. The default contracts supported by all four tools are listed as follows.

- For any object `o`, `o.equals(o)` returns true
- For any object `o`, `o.equals(null)` returns false

- For any objects $o_1$ and $o_2$, if $o_1$`.equals(`$o_2$`)`, then $o_1$`.hashCode() ==` $o_2$`.hashCode()`
- For any object `o`, `o.hashCode()` and `o.toString()` throw no exception

Randoop and Palus found the same number of bugs in all the open-source subjects, while Palulu and RecGen found less. Randoop did just as well as Palus because most of the bugs found in the subject programs are quite superficial. Exposing them does not need a specific method-call sequence or a particular value. For example, a large number of BCEL classes incorrectly implement the `toString()` method. For those classes, a runtime exception will be thrown when calling `toString()` on objects created by the default constructors. On the other hand, tests generated by Palulu are restricted by the inferred model, and thus miss some uncovered buggy methods.

The second part of the experiment evaluates Palus' bug finding ability using additional testing oracles written as JUnit theories. Since none of the authors is familiar with the subjects, we only wrote 5 simple theories for the Apache Commons Collections library based on our understanding. For example, according to the JDK specification, `Iterator.hasNext()` and all its overriding methods should never throw an exception. Thus, we wrote a theory for all classes that override the `Iterator.hasNext()` method. After that, we re-ran Palus to generate new tests, and Palus found one new bug. That is, the `hasNext()` method in `FilterListIterator` throws an exception in certain circumstances. This bug has been reported to, and confirmed by, the Apache Commons Collections developers.

*Palus at Google*

A slightly customized version of Palus (for Google's testing infrastructure) is internally used at Google. The source code of every Google product is required to be peer-reviewed, and goes through a rigorous testing process before being checked into the code base. We chose four products to evaluate Palus (Table 2.3). Each product has a comprehensive unit test suite. We executed the associated test suite to obtain a sample execution trace, then used Palus to generate tests. The results are shown in the bottom half of Table 2.4.

Palus revealed 22 previously-unknown bugs in the four products, which is more than Randoop and Palulu found within the same time using the same configurations. Each bug has been submitted with the generated test. All reported bugs had been confirmed. 8 of the reported bugs had been fixed.

Another 4 bugs were tracked down, and confirmed in the code of an internal code generation tool (that created the code Palus tested). The team building the code generation tool has been notified to fix them. Palus found those bugs in 5 hours of CPU time and 5 hours of human effort (spread among several days, including tool configuration and inspection of generated tests). In this case study, we only checked generated tests against default properties as listed in Section 2.5.5, since writing domain-specific testing oracles requires understanding of a specific product's code, which we do not have.

The bugs found by Palus eluded previous testing and peer review. The primary reason we identified is that for large-scale software of high complexity, it is difficult for testers to partition the input space in a way that ensure all important cases will be covered. Testers often miss corner cases. While Palus makes no guarantees about covering all relevant partitions, its combined static and dynamic test generation strategy permits it *learn* a call sequence model from existing tests, *fuzz* on specific legal method-call sequences, and then lead it to create tests for which no manual tests were written. As a result, Palus discovers many missed corner cases like testing for an empty collection, checking type compatibility before casting, and dereferencing a possibly null reference. As a comparison, the primary reason why Randoop and Palulu found fewer bugs is that they failed to construct good sequences for some classes under test due to argument constraints.

### *2.5.6 Experiment Discussion and Conclusion*

***Still uncovered branches.*** Palus achieves coverage far less than 100%. There are several reasons for this. First, the code under test often includes complex branches that are quite difficult to cover. As mentioned in Section 2.5.4, the code in Rhino's JavaScript parser and optimizer modules is only covered when provided a JavaScript program with certain structure. However, such an input is difficult to generate automatically. Using the `ParamValue` annotation described in Section 2.3.3 could help alleviate this problem. Second, the generated tests depend on the completeness of the inferred model. For parts of the code with constrained interfaces, if a sample trace does not cover that, it is difficult for Palus to create effective tests.

***Threats to validity.*** As with any empirical study, there are threats to the validity of our conclusions. The major threat is the degree to which the subject programs used in our experiment are representative

of true practice. Another threat is that we only compared Palus with three state-of-the-art test generation tools. To the best of our knowledge, these are the best publicly-available test generation tools for Java. Future work could compare with other tools, such as JCrasher [28].

***Analysis cost.*** Palus runs in a practical amount of time. Our evaluations were conducted on a 2.67GHz Intel® Core™2 PC with 12GB physical memory (3GB is allocated for the JVM), running Ubuntu 10.04 LTS. For the largest subject, Apache Commons, Palus recorded over 300MB of traces from executing its test suite. Palus finished building the enhanced call sequence graph and performing static analysis in 30 seconds. The performance has not been tuned, and we believe that it could be improved further.

***Applicability of different techniques.*** Feedback-directed random test generation (Randoop) is effective in generating tests for programs with few constraints. Palulu improves Randoop's effectiveness on programs with constrained interfaces, but may miss some states for program with dependence relations between methods. RecGen improves Randoop's effectiveness on programs with dependence relations between methods, but could fail to generate valid tests for programs with constrained interfaces. Palus improves on both sides. It may be most effective in generating test inputs for programs with constrained interfaces and inter-method dependences.

***Conclusion.*** The combined static and dynamic test generation technique used in Palus permits generating more effective tests, and finding more bugs in real-world code, compared to previous random test generation techniques.

## *2.6 Summary*

This chapter presents a combined static and dynamic automated test generation technique, and implements it in the Palus tool. The key idea of Palus is employing dynamic analysis to infer a software behavioral model (that captures the desired behaviors of the software under test) from execution traces and using the inferred model to guide random test generation. Thus, Palus could be regarded as *fuzzing on a specific legal path*. Integration with the JUnit theory framework permits developers to write project-specific testing oracles.

A comparison between four different test generation tools on six open-source programs and four Google products demonstrated the effectiveness of our approach. Our experience of applying Palus

on Google's code base suggests Palus can be effective in finding real-world bugs in large, well-tested products.

# Chapter 3

# EXPLAINING TEST FAILURES WITH DOCUMENTATION INFERENCE

The Palus technique described in Chapter 2 helps developers create tests, but it does not help developers interpret the test results. This chapter presents a technique, called FailureDoc, to assist developers in understanding a test failure. We first present the problem with illustrative examples in Section 3.1, then detail the FailureDoc technique in Section 3.2, and describe the empirical evaluation in Section 3.3.

## *3.1 Problem*

A failed test indicates a potential bug in the tested code. A developer must understand the cause of the failure and confirm its validity before starting bug-fixing. Recently, many automated test generation techniques [22, 104, 146, 148, 216] have been studied to create tests, but few techniques are proposed to *explain* why a test fails. Understanding *why a test fails* or even knowing *which part of the code should be inspected first in debugging* is a non-trivial task. This is a particular problem for automatically-generated tests which are often long and have poor readability, but it is also relevant for human-written tests.

For example, Figures 3.1 and 3.2 show a human-written test and an automatically-generated test, respectively. The first test in Figure 3.1 is associated with a JDK bug report. It shows that method `Arrays.toArray` is not type-safe. However, when executed, this test throws an `ArrayStoreException` at the last statement, which is not obviously related to any type safety issues. To confirm this reported bug, a developer must manually connect the available failure symptom (an `ArrayStoreException`) with possible failure causes. The second test, shown in Figure 3.2, is also not easy to understand. This automatically-generated test involves classes such as `ArrayList`, `TreeSet`, and `Collections` in the JDK, each of which contains hundreds of lines of code. Executing this test also does not give much useful information: the assertion simply fails without dumping any stack trace as debugging clues. Furthermore, the test has already been minimized: if any of

```
1. public void test1() {
2.   ArrayList<Number> nums = new ArrayList<Number>();
3.   Integer i = new Integer(1);
4.   boolean b0 = nums.add(i);
5.   Long l = new Long(-1);
6.   boolean b1 = nums.add(l);
7.   Integer[] is = new Integer[0];
8.   //This statement throws ArrayStoreException
9.   Integer[] ints = nums.toArray(is);
10. }
```

Figure 3.1: A human-written failed test. This test reveals a potential error in the JDK (bug id: 7023484).

```
1. public void test2() {
2.   int i = 1;
3.   ArrayList l = new ArrayList(i);
4.   Object o = new Object();
5.   boolean b0 = l.add(o);
6.   TreeSet t = new TreeSet(l);
7.   Set s = Collections.synchronizedSet(t);
8.   //This assertion (reflexivity of equals) fails
9.   assertTrue(s.equals(s));
10. }
```

Figure 3.2: An automatically-generated failed test. This test reveals an error in JDK version 1.6. It shows a short sequence of calls leading up to the creation of an object that is not equal to itself.

the calls is removed (and the code is fixed up so that the test compiles again), then the bug is not triggered, and the test passes.

Good documentation (i.e., code comments) can help developers quickly understand what source code does, facilitating program comprehension and software maintenance tasks [44]. Unfortunately, a human-written test is often poorly documented, and few automated test generation tools can adequately comment the code they generate. Even more importantly, when a test fails, the most useful documentation is relevant to the defect in the tested code and leads the developer to that defect. Later, when the test reveals a different defect, different documentation would be best. As a result of the lack of contextually-relevant documentation, developers must guess about what parts of the test and the tested code are relevant.

This chapter presents a fully-automated approach (and its tool implementation, called FailureDoc) to infer documentation for a failed test. FailureDoc is not an automated fault localization tool [97, 101, 228] that pinpoints the exact buggy code. Instead, it augments a failed test with

debugging clues: code comments that provide potentially useful facts about the failure, helping developers fix the bug quickly. Figures 3.3 and 3.4 show the inferred documentation for the failed tests of Figures 3.1 and 3.2, respectively.

In Figure 3.3, the comments above lines 5 and 6 reveal important clues that the test passes if object `l` is changed to `Integer` type or if `l` is not added to the `nums` list. These clues guide the developer to discover that the test failure is because the test erroneously adds two type-incompatible objects `i` and `l` into the list, and later casts both of them to `Integer` type. Such information is much more helpful in understanding why a test fails than merely dumping an `ArrayStoreException`. In Figure 3.4, the comment above line 4 discloses a crucial fact that the test passes if `Object o` implements `Comparable`. This clue guides the developer to inspect places where object `o` is used. In fact, the `TreeSet` constructor is buggy: it should not accept a list containing a non-`comparable` object, but it does. This indicates the exact cause and a possible bug fix.

To infer useful documentation, FailureDoc *simulates* developers' debugging activity. Its design is based on the following common debugging practice: given a failed test, a developer often tries to make some (minimal) edits to make it pass, observes the difference between passing and failing executions, then generalizes those failure-correcting edits to understand the failure cause. FailureDoc automates the above reasoning process, summarizing its observations as documentation.

FailureDoc works in four phases (Figure 3.5), namely *value replacement*, *execution observation*, *failure correlation*, and *documentation generation*. In the first phase, FailureDoc first generates an object pool, and then mimics developers' activity in correcting a failed test by repeatedly replacing existing values with possible alternatives. Each replacement creates a slightly mutated test. In the second phase, FailureDoc executes the mutated test, uses static slicing to prune irrelevant statements, and *selectively* observes its outcomes. In the third phase, FailureDoc uses a statistical algorithm to correlate the replaced values with their corresponding outcomes, identifying suspicious statements and their failure-correcting objects. In the final phase, for each identified suspicious statement, FailureDoc uses a Daikon-like technique [49] to summarize properties of the observed failure-correcting objects, translating them into explanatory code comments.

```
1. public void test1() {
2.   ArrayList<Number> nums = new ArrayList<Number>();
3.   Integer i = new Integer(1);
4.   boolean b0 = nums.add(i);
     //Test passes if line is: Integer l = new Integer(0);
5.   Long l = new Long(-1);
     //Test passes if l is not added to nums
6.   boolean b1 = nums.add(l);
7.   Integer[] is = new Integer[0];
8.   //This statement throws ArrayStoreException
9.   Integer[] ints = nums.toArray(is);
10. }
```

Figure 3.3: The failing test of Figure 3.1 with code comments inferred by the FailureDoc tool (highlighted by underline).

```
1. public void test2() {
2.   int i = 1;
3.   ArrayList l = new ArrayList(i);
     //Test passes if o implements Comparable
4.   Object o = new Object();
     //Test passes if o is not added to l
5.   boolean b0 = l.add(o);
6.   TreeSet t = new TreeSet(l);
7.   Set s = Collections.synchronizedSet(t);
8.   //This assertion (reflexivity of equals) fails
9.   assertTrue(s.equals(s));
10. }
```

Figure 3.4: The failing test of Figure 3.2 with code comments inferred by the FailureDoc tool (highlighted by underline).

### *3.2 Technique*

Figure 3.5 illustrates the workflow of FailureDoc with a simple example. FailureDoc consists of four major modules, working in a pipelined manner.

***(1) Value Replacement***. This module takes a failed test as input. It first uses a randomized algorithm to create an object pool containing instances of all needed classes, then mutates the failed test by repeatedly replacing expressions in the test code with possible alternatives from the created object pool to construct a set of slightly mutated tests (Section 3.2.1).

***(2) Execution Observation***. This module executes the mutated tests to obtain execution traces. For each mutated test, this module uses static slicing to prune all irrelevant statements from the execution trace (Section 3.2.2).

***(3) Failure Correlation***. This module takes as inputs the replaced values and the observed

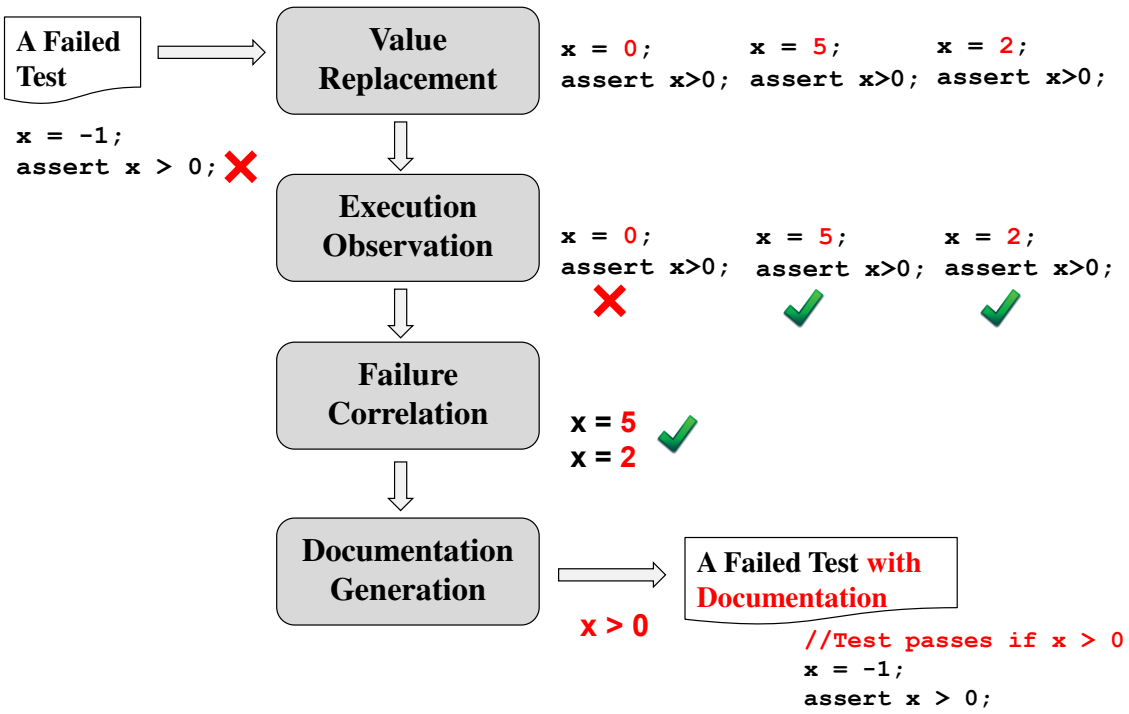



Figure 3.5: Illustration of FailureDoc's workflow with a simple example. FailureDoc augments a failed test with explanatory documentation (debugging clues) in four steps.

outcomes. It uses a statistical algorithm to identify a small set of suspicious statements (that have a strong correlation with the test failure) and their *failure-correcting* objects (Section 3.2.3).

***(4) Documentation Generation.*** This module generalizes properties of the observed failure-correcting objects for each suspicious statement, then converts the generalized properties into documentation (Section 3.2.4).

### *3.2.1 Value Replacement*

Given a failed test, the value replacement module repeatedly replaces expressions in the test code with possible alternatives to construct a set of mutated tests. For example, in Figure 3.2, FailureDoc can replace `int i = 1` on line 2 with `int i = 0`, or replace `TreeSet t = new TreeSet(1)` on line 6 with `TreeSet t = new TreeSet(); t.put(10);`.

To correctly and efficiently implement value replacement, two key challenges must be addressed.

First, FailureDoc must create new values to replace an existing one. Second, for the sake of efficiency, FailureDoc must select among the new values it creates for each existing value.

*Value Generation*

The FailureDoc technique can be instantiated using any value generation technique, such as exhaustive generation up to a given size bound [22]. Our current implementation uses an existing random test generation algorithm, that of Randoop [148]. We now briefly describe it.

FailureDoc parses the failed test to extract all referred-to classes. For example, the failed test in Figure 3.2 refers to classes `Integer`, `Object`, `ArrayList`, `TreeSet`, `Set`, and `Collections`. Optionally, the user can also provide FailureDoc additional classes for value generation, but our experiments did not use this capability.

FailureDoc next uses an existing random test generation algorithm [146, 148] to create object instances of the referred-to classes, and keeps all created objects in a value pool. The algorithm iteratively builds a method-call sequence that produces an object value, by randomly selecting a method or constructor to invoke, using previously computed values as inputs. FailureDoc uses object values in the pool to mutate the failed test. The mutated tests are never shown to the user.

*Value Selection*

After generating an object pool, a natural question is how to select possible alternatives for each expression. Many of the objects in the pool may be similar. Hence, the naive approach of choosing every type-compatible object from the pool to replace the existing value is unnecessary and inefficient. On the other hand, understanding the failure cause requires selecting a group of diverse objects as replacement candidates, in order to expose different test behaviors. Randomly selecting a group of objects from the pool may end up with the same test behavior, which would not be helpful to infer useful information.

To alleviate this problem, FailureDoc *adaptively* selects a diverse set of objects as replacement candidates based on an abstract object profile representation (described below). Adaptive selection reduces the likelihood of choosing a group of similar objects, in which most of the objects reveal the same test behavior while failing to reveal other behaviors.

For each expression $e$ of type $T_e$ in the test code, FailureDoc creates approximately $k$ mutants ($k$ is user-settable; our experiments used the default value $k$ = 20) by replacing $e$ by another value from the created pool. If the pool has $k$ or fewer elements of type $T_e$, FailureDoc uses them all to create mutants. Otherwise, FailureDoc tries to choose $k$ values that are as different from one another as possible, in the following way.

For each object of type $T_e$ in the pool, FailureDoc computes its abstract object profile. Let $a$ be the number of distinct abstract object profiles. FailureDoc randomly chooses $\lceil k/a \rceil$ values with each abstract object profile.

An *abstract object profile* is a boolean vector that abstracts the object's concrete state (the values of its fields). Each field of the object maps to a distinct set of boolean values in the vector.

- A concrete numerical value $v$ (of type `int`, `float`, etc.), maps to three abstract values $v < 0$, $v = 0$, and $v > 0$.

- A concrete boolean field value $v$ maps to two abstract values $v =$ `true` and $v =$ `false`.

- A concrete enumeration value $v$ chosen from choices $e_1, \ldots, e_j$ maps to $j$ abstract values, $v = e_i$ for each $1 \leq i \leq j$.

- A concrete object reference value $v$ is mapped to two abstract values $v =$ `null` and $v \neq$ `null`.

- A concrete array or collection value $v$ is mapped to three abstract values as follows: the same two abstract values as for objects (since an array or collection is an object); one abstract value for whether $v$ is empty.

For example, suppose the `TreeMap` class has two fields `int size` and `Set entrySet`, which represent the map size and the internal data representation, respectively. The following table summarizes the corresponding abstract object profiles for an empty and a non-empty `TreeMap` object. **T** means the property holds.

| A `TreeMap` object | Abstract Object Profile | | | | | |
|---|---|---|---|---|---|---|
| | `size` | | | `entrySet` | | |
| | $< 0$ | $= 0$ | $> 0$ | $=$ `null` | $\neq$ `null` | is empty? |
| an empty map | F | **T** | F | F | **T** | **T** |
| a non-empty map | F | F | **T** | F | **T** | F |

The above abstract object profile, which is used in our implementation, looks at the top level of the concrete representation. The abstract object profile can go as deep in the object as desired. It is straightforward to extend it to more of the concrete representation, as follows:

- Enrich the abstraction of a concrete object reference or enumeration value, by adding abstract values for each of its fields. This corresponds to following two field references: the abstract object profile depends not just on `x.f`, but also on `x.f.g`.

- Enrich the abstraction for arrays/collections, by adding existential abstract values according to the type of its elements. For example, for an array of objects, add one abstract value indicating whether the array contains any `null` element, and one abstract value indicating whether the array contains any non-`null` element.

### *3.2.2 Execution Observation*

FailureDoc next executes the mutated tests, observing their execution outcomes. For each mutated test, FailureDoc records two types of information:

- the test execution outcome

- the value computed by each expression

FailureDoc classifies the test execution outcome into three categories: *pass*, *fail*, and *unexpected exception*. The first two categories represent that a mutated test throws no exception or throws the

exact same exception as the original failing test. The *unexpected exception* category represents the scenario that a different uncaught exception is thrown and the test aborts without producing a final result. For example, if FailureDoc replaces `int i = 1` with `int i = -1` at line 2 of Figure 3.2, an `IllegalArgumentException` will be thrown at line 3 when executing the mutated test, since `ArrayList` requires a non-negative integer as input to its constructor.

When recording expression values in a mutated test, an important problem is which expression values FailureDoc should record. A straightforward approach is to record the runtime values of all expressions in the mutated test. However, the replaced value may *mask* the effects of some previously-created values. So recording them would introduce incorrect noisy data for the follow-up failure correlation phase. For example, if FailureDoc replaces `TreeSet t = new TreeSet(l)` on line 6 of Figure 3.2 with `TreeSet t = new TreeSet()`, then objects `i`, `l`, and `o` created on line 2–5 will never affect the test execution result. Those object values should not be correlated with the test execution result. Therefore, to correctly observe expression values, such *masking* effects must be identified.

To address this problem, for each mutated test, FailureDoc computes a static backward slice [206] from the assertion statement that fails in the original test to identify expressions whose outcomes may affect the test execution result. Then, FailureDoc only records computed values of those identified expressions. For example, when executing a mutated test created by replacing `TreeSet t = new TreeSet(l)` on line 6 in Figure 3.2 with `TreeSet t = new TreeSet()`, FailureDoc analyzes the mutated test code and identifies that only the expressions on lines 6 and 7 may affect the assertion on line 9.

FailureDoc records the execution outcomes of a mutated test in a vector $V = \langle v_1, \ldots, v_n \rangle$, where $n$ is the number of statements in the test. If the $i$th statement is not in the backward slice from the assertion, then $v_i = Ignore$ ($Ignore$ means the outcome is not recorded). Otherwise, $v_i$ is the outcome object of the $i$th statement.

After executing all mutated tests, FailureDoc collects a set of outcome vectors. These will be used as input to the failure correlation module, which identifies suspicious statements that are most likely to cause the failure.

### *3.2.3 Failure Correlation*

We devised an offline statistical algorithm that isolates a small set of suspicious statements in a failed test. Our algorithm is a variant of a well-established cooperative bug isolation technique [121]. The basic idea is to identify likely buggy statements by correlating the observed values with the execution results in a set of passing and failing executions.

However, the statistical algorithm described in [121] cannot be directly applied to our problem domain, for two reasons. (1) The original algorithm uses the boolean value of an instrumented predicate as the feature vector to identify likely buggy *predicates*, while FailureDoc needs to use multiple observed values (in an outcome vector) to isolate suspicious *statements*. (2) Merely identifying suspicious statements is not sufficient for FailureDoc. FailureDoc also needs to associate each identified statement with a set of *failure-correcting* objects, using any of which to replace the existing value will make the test pass. The properties of such *failure-correcting* objects will be generalized and translated into human-readable comments by the follow-on documentation generation module.

To address the above two limitations, we first define a new metric $Pass$ and re-define three existing metrics $Context$, $Increase$, and $Importance$ as proposed in [121], then present a new statistical algorithm at the end of this section. Interested readers can refer to [121] for more details on the design of the three existing metrics.

For the $i$th statement $s_i$ in a failed test, there are $j$ different replacing values recorded in outcome vectors, which we denote as $v_{i1}$, $v_{i2}$, ..., $v_{ij}$. For statement $s_i$, we define its *failure-correcting* object set $FC_i = \{v_{ik} \mid \text{using } v_{ik} \text{ to replace the existing value makes the test pass}\}$.

Let $Pr(A|B)$ denote the conditional probability of the event $A$ given event $B$. Let $S(v_{ij})$ be the number of successful runs in which value $v_{ij}$ replaces the $i$th statement, and let $F(v_{ij})$ be the number of failing runs in which value $v_{ij}$ replaces the $i$th statement. The probability of the test passing when using $v_{ij}$ to replace the existing value in the $i$th statement is:

$$Pass(v_{ij}) = \frac{S(v_{ij})}{S(v_{ij}) + F(v_{ij})}$$

A lower score shows weaker correlation between $v_{ij}$ and the test failure. However, a single metric is often insufficient to identify suspicious code [121], which we also experimentally verified

in Section 3.3.1 for our problem domain. The major reason is that $Pass(v_{ij})$ does not consider the number of available executions and code execution context. Intuitively, for two replacing values $v_{i1}$ and $v_{i2}$, if $Pass(v_{i1}) = Pass(v_{i2})$ but $v_{i1}$ is observed in more executions, we should have a stronger belief that $v_{i1}$ is correlated with the failure. Additionally, the observation of $v_{ij}$ could be affected by its execution context. In some cases, high $Pass(v_{ij})$ does not necessarily mean $v_{ij}$ is an interesting *failure-correcting* object, it is possible that the decision that eventually causes the test to pass is made earlier, and the high $Pass(v_{ij})$ score just reflects the fact that this value is observed after the decision has been made.

For those reasons, we re-define metrics $Context$, $Increase$, and $Importance$ by considering the code execution context and number of available executions, as follows.

$Context(v_{ij}) = \Pr(\text{Test Passes} \mid \text{the ith line is observed})$

In the above definition, $Context(v_{ij})$ is the probability that the output value of the $i$th line is recorded (has not been pruned after performing slicing in Section 3.2.2) in a passing execution. We define $Context(v_{ij})$ as:

$$Context(v_{ij}) = \frac{SR(i)}{SR(i) + FR(i)}$$

In the above definition, $SR(i)$ is the number of passing executions when the output of the $i$th statement is recorded, and $FR(i)$ is the number of failing executions when the output of the $i$th statement is recorded. Then, we re-define $Increase(v_{ij})$ as:

$$Increase(v_{ij}) = Pass(v_{ij}) - Context(v_{ij})$$

A value $v_{ij}$ with $Increase(v_{ij}) \leq 0$ is unlikely to be a *failure-correcting* object and will be discarded by FailureDoc.

We finally re-define the $Importance$ metric. It considers the number of available passing executions $S(v_{ij})$, to avoid only representing special cases (observed from a small number of executions) but not being applicable to more general cases.

$$Importance(v_{ij}) = \frac{2}{1/Increase(v_{ij}) + 1/\log(S(v_{ij}))}$$

**Input**: a failed test $t$
**Output**: a set of suspicious statements $Stmts$, each of which is associated with a set of failure-correcting objects

1: $Stmts \leftarrow \emptyset$
2: **for** each statement $s_i$ in $t$ **do**
3: $\quad FC_i \leftarrow \{v_{ij} \mid S(v_{ij}) > 0 \wedge F(v_{ij}) = 0 \ \wedge Increase(v_{ij}) > 0 \wedge Importance(v_{ij}) > k\}$
4: $\quad$ **if** $FC_i \neq \emptyset$ **then**
5: $\quad\quad Stmts \leftarrow Stmts \cup \langle s_i, FC_i \rangle$
6: $\quad$ **end if**
7: **end for**
8: **return** $Stmts$

Figure 3.6: Algorithm for isolating a set of suspicious statements. Each isolated statement is associated with a failure-correcting object set. The threshold $k$ for the $Importance$ metric is user-settable; our experiments use its default value 0.8.

Figure 3.6 shows our algorithm for isolating suspicious statements. A statement is classified as suspicious iff its failure-correcting object set $FC \neq \emptyset$.

### *3.2.4 Documentation Generation*

For each identified suspicious statement, FailureDoc uses a Daikon-like [49] technique to summarize common properties of its *failure-correcting* objects, and convert the summarized properties into human-readable comments. Each comment indicates a different way to cause the failed test to pass. Thus, even if the comments provide different information, the comments are never in conflict with one another.

#### *Object Property Generalization*

FailureDoc reports properties that are true over all failure-correcting values. The essential idea is to use a generate-and-check algorithm to test a set of pre-defined potential properties against the values, and then report those properties that are not falsified.

Given a set of objects $FC = \{o_1, o_2, \ldots, o_n\}$, FailureDoc checks the following properties:

- **Type:** do all objects in $FC$ have the same type? For example, are all objects `Comparable`, or of `Collection` type?

| Program (version) | Program size | | | Failed tests | | Commented tests | | Execution details | |
|---|---|---|---|---|---|---|---|---|---|
| | LOC | Classes | Methods | Tests | Statements | Tests | Comments | Mutants | Time |
| Time And Money (0.51) | 2372 | 29 | 492 | 2 | 81 | 1 | 3 | 993 | 114 |
| Apache Commons Primitives (1.0) | 9368 | 210 | 1739 | 2 | 150 | 2 | 3 | 4740 | 1079 |
| Apache Commons Math (2.2) | 14469 | 131 | 1333 | 3 | 144 | 2 | 13 | 2037 | 271 |
| Apache Commons Collections (3.2.1) | 55400 | 445 | 5350 | 3 | 83 | 3 | 6 | 1987 | 780 |
| `java.util` package (1.6.0_12) | 48026 | 191 | 3387 | 2 | 27 | 2 | 4 | 734 | 29 |
| Total | 129635 | 1006 | 12301 | 12 | 485 | 10 | 29 | 10491 | 2273 |

Figure 3.7: Subject programs and experimental results used in evaluating FailureDoc. Column "LOC" is the number of lines of code, as counted by LOCC [127]. Column "Commented tests" is the number of failed tests for which FailureDoc can infer documentation. Column "Comments" is the total number of inferred comments for the documented tests (one comment per suspicious statement). "Mutants" is the total number of mutated tests created by the value replacement module (Section 3.2.1). Column "Time" is the total time (in seconds) used in the whole documentation inference process, including the time spent on tests for which FailureDoc cannot infer documentation.

- **Abstract Object Profile:** do all objects in $FC$ have the same abstract object profile (Section 3.2.1)?

Like the properties in Daikon [49], the properties in FailureDoc can be viewed as forming a lattice based on subsumption (logical implication). The FailureDoc implementation takes advantage of these relationships in order to improve the intelligibility of the output. For example, if all objects in $FC$ are both integer type and comparable, FailureDoc will suppress the weaker property: being comparable is logically implied by being of integer type.

We produced the list of properties in the abstract object profile by proposing a basic set that seemed natural and generally applicable, based on our debugging experience. We later added other properties we found useful in revealing debugging clues (e.g., checking whether a collection object is empty or not). Compared to the invariant detection engine implemented in Daikon [49], the property set checked by FailureDoc is highly tailored for the debugging purpose. It discards many scalar properties that are extensively used in Daikon, and adds some properties that Daikon lacks, such as checking whether all objects have the same abstract object profile.

Consider the example in Figure 3.2, and suppose the failure correlation module identifies line 4 as suspicious. FailureDoc will take its failure-correcting object set as inputs, such as { `"hi"`, `100`, (**`byte`**)`1`}, and summarize the property that all objects are `Comparable`.

*Documentation Summarization*

FailureDoc concatenates all reported properties and converts them into descriptive documentation. It employs a small number of simple translations to phrase common Java idioms. The translated documentation is presented like a description of *property* as follows:

- `x = null` **becomes** "x is set to: `null`"
- `x == (Integer)0` **becomes** "the line is: `Integer x = new Integer(0);`" (when the type of `x` is not `Integer` in the failed test)
- `x instanceof Comparable` **becomes** "x implements `Comparable`"
- `x.add(o)` returns `false` **becomes** "`o` is not added to `x`" (when `x` is a `Collection` type object)
- `x` has an abstract profile `TreeMap`⟨`size = 0, entrySet`≠`null`⟩ **becomes** `x` is a `TreeMap` object, in which `size` is `0` and `entrySet` is not `null`

As shown in Figure 3.4, the translated documentation (code comment) is presented in the form of: "Test passes if *property*". If multiple properties are reported, FailureDoc generates documentation in the form of "Test passes if *property 1* or *property 2*".

### *3.3 Evaluation*

We have implemented a tool, called FailureDoc, and investigated the following two research questions:

1. **RQ1:** Can FailureDoc generate meaningful documentation for failed tests from realistic programs?
2. **RQ2:** Does the inferred documentation help developers to understand failed tests?

To answer these research questions, we designed an experiment and a controlled user study.

1) For **RQ1**, we applied FailureDoc to five real-world programs to infer explanatory documentation for failed tests (Section 3.3.1). We examined the inferred documentation and also sent it to the subject developers for feedback.

2) For **RQ2**, we designed a controlled user study to investigate whether the generated documentation aids bug diagnosis (Section 3.3.2). The user study involved 16 participants with an average of 4.1 years of Java programming experience.

### *3.3.1 Experiment: Inferring Debugging Clues*

FailureDoc takes as input a failed test. We generated failed tests by running Randoop [148] on five real-world subject programs (Apache Commons Collections [4], Primitives [6], Math [5], Time and Money [194], and `java.util` [94]. Randoop checked 5 default programming rules defined in Java such as the symmetry property of equality: `o.equals(o)`. Randoop reported 12 failed tests; each one indicates a distinct, real bug in a subject program. On average, each test has 41 lines of code excluding assert statements. Next, FailureDoc inferred documentation for each failed test. Finally, we examined FailureDoc's output to judge the quality of inferred documentation. Figure 3.7 summarizes the experimental results.

***Results.*** Like the example in Figure 3.2, most failed tests involve complex code interactions between multiple classes. It was hard for us to tell the failure cause by simply looking at the source code or executing the test.

FailureDoc successfully inferred documentation for 10 of 12 failed tests. The comments FailureDoc creates are reasonably succinct, approximately 1 comment per 17 lines of test code. FailureDoc is fast enough for practical use, taking 189 seconds on average to infer documentation for one failed test.[1] The time used to infer documentation for a test is roughly proportional to the length of the test, rather than the size of the tested program. Most of the time is spent performing value replacement, because reflectively executing a failed test with multiple input values (in Java) takes a considerable amount of time.

---

[1]The experiment used a 2.67GHz Intel(R) Core 2 PC with 2GB physical memory (512MB is allocated for the JVM), running Fedora 12.

```
public void testListOrderedSet() throws Throwable {
1.  ListOrderedSet listOrderedSet0 = new ListOrderedSet();
    ...
23. ListOrderedSet listOrderedSet9
      = ListOrderedSet.decorate((Set)listOrderedSet0, list6);
    ...
24. Integer i2 = new Integer(0);
25. boolean b3 = listOrderedSet0.add((Object)i2);
    //Test passes if line is: Integer s0 = new Integer(0);
26. Short s0 = new Short((short)1);
    //Test passes if s0 is not added to listOrderedSet0
27. boolean b4 = listOrderedSet0.add((Object)s0);
    ...
33. ListOrderedSet listOrderedSet11 = new ListOrderedSet();
    ...
58. int i5 = listOrderedSet21.size();
    //Test passes if i5 is not added to listOrderedSet11
59. boolean b8 = listOrderedSet11.add((Object)i5);
    ...
    //this assertion (transitivity of equals) fails
61. assertTrue(listOrderedSet11.equals(listOrderedSet9)
      == listOrderedSet9.equals(listOrderedSet11));
}
```

Figure 3.8: A failed test for Apache Commons Collections. FailureDoc automatically augmented it with the underlined debugging comments.

***Example.*** Figure 3.8 shows a documented test from subject Apache Commons Collections. The comments indicate which part of the test code developers should inspect first, while ignoring other irrelevant method calls and variables.

In Figure 3.8, the comments indicate three ways to correct this failed test by either changing the value of `s0` on line 26 from `(Short)1` to `(Integer)0`, or not adding `s0` to `listOrderedSet0` on line 27, or not adding `i5` to `listOrderedSet11` on line 59. In particular, the comment above line 26 reminds developers the test will pass if an existing value is added to `listOrderedSet0` (the same value as the `new Integer(0)` object defined on line 24, is added to `listOrderedSet0` on line 25). This information guides developers to check the code inside method `ListOrderedSet.add(Object)`. After further inspection, developers would find the `add` method does not update a `ListOrderedSet` object state correctly when adding a new element: it only adds the element to the `collection` field, but forgets to update the `setOrder` field. Note that `listOrderedSet9` is just a wrapping object of `listOrderedSet0` on line 23. Thus, updating `listOrderedSet0` also changes the state of `listOrderedSet9`. Thus, this comment indicates the exact cause for the assertion failure.

FailureDoc failed to infer comments for two failed tests, in subjects Time And Money and Apache

Commons Math. That is mainly due to the characteristics of the revealed bugs. In Time And Money, the static factory method `everFrom` in class `TimeInterval` incorrectly passes a *hard-coded* `null` value to a constructor. In subject Apache Commons Math, the revealed bug is due to the fact that two objects `new Double(0.0d)` and `new Double(-0.0d)`, though numerically equal, have different hash codes. For both failed tests, there is no way for client code to use value replacement to correct them. Therefore, FailureDoc fails to infer useful code documentation.

***Feedback from Developers.*** We sent the documented failed tests to the subject program developers, asking them to judge the documentation quality. We received several pieces of positive feedback from the developers. Luc Maisonobe, a developer of Apache Commons Math gave us the following comment:

> *I think these comments are helpful. They give a hint about what to look at. In both (failed test) cases, I had to run the test in a debugger to see exactly what happened but the comment showed me exactly the variable to look at.*

All reported bugs and documented tests are publicly available at: `http://www.cs.washington.edu/homes/szhang/failuredoc/bugreports/`.

***Experimental comparison with delta debugging.*** Delta debugging [228] is a general technique to isolate failure-inducing inputs. It has the potential to isolate suspicious statements from a failed test, just as our statistical algorithm (Section 3.2.3) does.

We implemented the isolating delta debugging algorithm described in [228], and applied it to the 12 failed tests in Figure 3.7. Delta debugging [228] experimentally validates whether a certain statement is suspicious, by removing it from the failed test code and re-executing the simplified test, to see whether the same failure occurs again. One limitation of delta debugging is that it cannot isolate statements whose removal would cause a compilation error.

Delta debugging isolated 4 suspicious statements in 3 failed tests (such as, statement 5 in Figure 3.4, and statements 27 and 59 in Figure 3.8). By comparison, our statistical algorithm isolated 29 suspicious statements in 10 failed tests, including all 4 statements isolated by delta debugging.

Even more importantly, the statements isolated by delta debugging gave limited information for understanding the failure. For instance, in Figure 3.4, the comment above statement 5 indicates

| Goal | Success Rate | | Time Used (in minutes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | JUnit | FailureDoc | JUnit | | | | FailureDoc | | | |
| | | | mean | sd | max | min | mean | sd | max | min |
| Understand Failure | 75% | 75% | 22.6 | 7.4 | 30 | 11 | 19.9 | 10.0 | 30 | 2 |
| Understand Failure + Fix Defect | 35% | 35% | 27.5 | 4.8 | 30 | 17 | 26.9 | 6.9 | 30 | 5 |

(a) Comparison of original un-documented failed tests (column "JUnit") with FailureDoc-documented tests (column "FailureDoc").

| Goal | Success Rate | | Time Used (in minutes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Delta debugging | FailureDoc | Delta debugging | | | | FailureDoc | | | |
| | | | mean | sd | max | min | mean | sd | max | min |
| Understand Failure | 75% | 75% | 21.7 | 8.1 | 30 | 6 | 20.0 | 9.6 | 30 | 2 |
| Understand Failure + Fix Defect | 40% | 45% | 26.1 | 6.1 | 30 | 9 | 26.5 | 6.8 | 30 | 5 |

(b) Comparison of tests annotated with faulty statements isolated by Delta debugging [228] (column "Delta debugging") with FailureDoc-documented tests (column "FailureDoc").

Figure 3.9: User study results. "Success Rate" represents the percentage of finished tests for a certain goal. "Time Used (in minutes)" represents the average time to complete one task. We used 30 minutes (the maximum allowed time) for participants who failed to complete a certain goal.

that the `Object o` defined on line 4 is related to the failure, but does not tell *why the test fails*. In contrast, FailureDoc additionally isolates statement 4 as suspicious, and generates a comment for it, guiding developers to better understand the failure cause.

Section 3.3.2 empirically compares delta debugging with FailureDoc via a user study to show that delta debugging produces less useful results in understanding the failure cause.

***Experimental comparison with a single metric statistical algorithm.*** We next verified the necessity of using multiple metrics ($Pass$, $Increase$, and $Importance$ defined in Section 3.2.3) in isolating suspicious statements. We implemented a simpler statistical algorithm only using metric $Pass$, and compared its results with FailureDoc's output. This simpler algorithm changes line 3 of Figure 3.6 to

$$FC_i \leftarrow \{v_{ij} \mid Pass(v_{ij}) > 0 \wedge F(v_{ij}) = 0\}$$

This simpler algorithm isolated 32 suspicious statements from the 10 failed tests in Figure 3.7: the 29 suspicious statements isolated by our algorithm, plus 3 additional statements that are implied by FailureDoc's output. The 3 additional statements are further from the root cause, and are less useful in understanding the test behavior.

### 3.3.2 User Study: Understanding Failed Tests

We performed a controlled user study to investigate two questions. First, can FailureDoc help developers understand a failed test and fix the revealed bug? Second, is the information provided by FailureDoc more useful than delta debugging [228], a state-of-the-art automated debugging technique?

The participants were given test cases and asked to understand and fix the underlying error. There were three experimental treatments: some test cases were as produced by Randoop; some had been annotated with suspicious statements identified by delta debugging; and some had been annotated with FailureDoc documentation.

***Setup.*** The participants were 16 graduate students in computer science. On average, they had 4.1 years of Java programming experience (min = 1, max = 7, sd = 1.7), and 1.9 years of JUnit experience (min = 0.1, max = 4, sd = 1.5). None of them was familiar with the subject code. Before the user study started, we gave each participant a 15-minute tutorial about the basic concept of JUnit tests, the default contracts that Randoop's JUnit tests check, and (when relevant) the format of FailureDoc's inferred documentation.

Each participant was given 30 minutes per failed test case. For each failed test, we gave the participant two goals. First, participants were asked to *understand* why the test fails, write down its failure cause, and tell us when they had found it. Second, participants were asked to write a patch to *fix* the defect. After all participants finished their tasks, we checked the correctness of the identified failure cause (the first goal) and the proposed bug fixes (the second goal).

Each participant was assigned to understand and fix 2–4 different failed test cases chosen from Figure 3.7. 5 participants received two test cases with FailureDoc documentation, then two un-documented test cases. 5 participants received first two un-documented test cases, then two FailureDoc-documented test cases. 3 participants received 1 FailureDoc-documented test case, then 1 test case annotated with suspicious statements identified by delta debugging. 3 participants received 1 delta-debugging-annotated test case, then 1 FailureDoc-documented test case. We assigned fewer participants to the delta debugging treatment, because delta debugging identified suspicious statements for only 3 failed tests. For the other 7 failed tests, we re-used the experimental data for the un-documented test cases. Using this approach, each subtable of Figure 3.9 compares the same

experimental subjects, which avoids conflating individual differences with treatment differences.

***Results.*** Figure 3.9 summarizes the user study results.

As shown in Figure 3.9(a), participants using FailureDoc-documented tests and un-documented tests completed the same number of tasks. However, participants using FailureDoc-documented tests understood the failure cause 14% faster (2.7 minutes faster) than the participants using un-documented tests. For each defect, participants with FailureDoc-inferred documentation spent slightly less time (0.6 minute) to fix, than participants without the documentation. This indicates that the explanatory documentation is most useful for understanding the failure cause.

As shown in Figure 3.9(b), participants using FailureDoc-documented tests completed more tasks than participants with the aid of delta debugging (40% versus 45% success rate in fixing defects). The two treatments led to the same success rate in understanding a failed test, but participants using FailureDoc-documented tests spent 8.5% less time (1.7 minutes) to do so. The delta debugging treatment led to slightly faster fixes than with FailureDoc (0.4 minute), despite delta debugging's lower success rate. This is primarily due to one developer who fixed one defect extremely quickly under the delta debugging treatment. In retrospect, setting the maximum permitted time to be larger would have yielded more discriminating results: the average *successful* bug fix (across all treatments) took 23.2 minutes, which is not much less than the 30 minutes that the statistics use for unsuccessful participants.

***Participants' feedback.*** After the user study, we asked all participants to complete a survey, writing down their feedback and suggestions on the inferred documentation. 15 out of 16 participants thought unfamiliarity with the subject code was the major reason for their slow progress or their failure to fix the defect. When asked to classify the usefulness of inferred documentation as either *very useful*, *useful*, *not very useful*, and *not useful at all (misleading)*, 1 participant thought the documentation was *very useful*, 12 out of 16 participants thought the documentation was *useful*, while the remaining 3 participants thought the documentation was *not very useful*.

The 13 participants who found the documentation useful (or very useful) have an average of 4.3 years of Java programming experience, while the remaining 3 participants have an average of 2.6 years of experience. Two participants said they expected the comments to exactly pinpoint the buggy code (that is not the goal of FailureDoc), and thus thought the inferred documentation less useful, and

the other participant said the comments were not very useful if he was not familiar with the tested program. More experienced participants could leverage their own experience with the given hint to find the right code to inspect.

During the study, one participant accidentally overlooked a comment in the assigned test, and spent over 25 minutes in understanding one failed test. However, as soon as he noticed the overlooked comment, he understood why the test fails. That participant gave us the following comment: *I should have noticed the comments (earlier). The comment at line 68 did provide information very close to the bug!*

In contrast, another participant efficiently leveraged the inferred documentation, understood the failure cause, and proposed a good fix for the failed test in less than 5 minutes. He gave us the following comment: *The comments are useful, because they indicate which variables are suspicious, and help me narrow the search space.* The participant who thought the comment was very useful said: *this kind of comments can help to eliminate the wrong search path and the possibility of missing a bug, thus reducing the debugging time*.

The three participants who thought the comments were not very useful said the information provided by FailureDoc interfered with their established debugging habits. They gave three pieces of negative comments. One participant said: *the comments usually provide information about data and control dependencies, but their meanings are not so obvious for bug fixing. They can be more useful, if they are more descriptive in natural language*. Another participant said that his assigned test was already very simple, and though the comments helped him understand what the test was doing, they were not so useful. The third participant said: *the comments, though [they] give useful information, can easily be misunderstood, when I am not familiar with the [program].*

After the user study, we asked the 6 participants who received both FailureDoc-documented tests and tests annotated with suspicious faulty statements by delta debugging to judge which information is more useful. All 6 participants thought FailureDoc provided richer and more useful information than delta debugging, since the documentation not only indicated which statements are relevant to the bug, but also gave hints on why the test failed.

***Threats to validity.*** There are three major threats to validity. First, the five programs and the diagnosed bugs may not be representative. Thus, we cannot claim the results can be extended to an

arbitrary program. Second, the generality of our user study is obviously limited: this was a small task, a small sample of people, limited time, and unfamiliar code. Third, the differences in the means are relatively small and not statistically significant, in part because of the 30-minute time limit. These three threats can be reduced by performing experiments on more subjects and users.

***Experimental conclusions.*** We have three chief findings: (1) compared to an undocumented test, FailureDoc's inferred documentation slightly speeds up the task of understanding and fixing a bug, (2) compared to delta debugging, FailureDoc slightly speeds up the task of understanding a bug, and leads to greater success in fixing the bug, and (3) the inferred documentation is more useful for more experienced developers.

### *3.4 Summary*

This chapter has presented FailureDoc, a fully-automated technique to infer documentation to explain failed tests. FailureDoc uses a combination of several lightweight techniques to achieve its goal. The key idea of FailureDoc is leveraging automated test generation to create additional failure-relevant executions (i.e., passing tests that are slightly different from a given failing test) and using statistical analysis to correlate the execution difference to certain program properties. In our experiment, FailureDoc successfully inferred documentation for 10 out of 12 failed tests from five real-world programs, showing good scalability. The documentation inferred by FailureDoc revealed important information about the test failure, guiding developers to inspect the right code place. Our user study and developers' reaction further demonstrated its usefulness.

# Chapter 4

# DIAGNOSING CONFIGURATION ERRORS IN COMPLEX SOFTWARE SYSTEMS

Palus and FailureDoc help developers create better test suites and understand test results, but they do not help users diagnose software errors. As the complexity of modern software increases, software end-users frequently encounter tasks that they do not know how to perform, such as how to configure a complex software system or how to adapt to software features. To address this challenge, the following chapters (Chapters 4 to 6) present three program-analysis-based techniques to help software end-users diagnose software errors.

This chapter presents a technique, called ConfDiagnoser, to assist software end-users in troubleshooting software configuration errors. Section 4.1 describes the configuration error diagnosis problem, Section 4.2 gives a real-world configuration error and shows how ConfDiagnoser helps diagnose this error, Section 4.3 details the ConfDiagnoser technique, Section 4.4 describes our tool implementation, Section 4.5 presents the evaluation, and Section 4.6 concludes.

## *4.1 Problem*

Many modern software systems support a range of configuration options for users to customize their behavior. This flexibility has a cost: a small configuration error may cause hard-to-diagnose behavior.

Software configuration errors are errors in which the software code and the input are correct, but the an incorrect value is used for a configuration option so that the software does not behave as desired. Such errors may lead the software to crash, produce erroneous output, or simply perform poorly. Even when an application outputs an error message, it is often cryptic or misleading [14,87,219,222]. Users may not even think of configuration as a cause of their problem. In practice, software configuration errors are *prevalent*, *severe*, and *hard to debug*, but they are *actionable* for users to fix.

**Prevalent.** A recent analysis of Yahoo's mission-critical Zookeeper service showed that software

misconfigurations accounted for the majority of all user-visible failures [178]. Configuration-related issues caused about 31% of all failures at a commercial storage company [222].

**Severe.** Configuration errors can have disastrous impacts. For example, an outage in Facebook due to an incorrect configuration value left the website inaccessible for about 2 hours [53]. The entire `.se` domain of Sweden was unavailable for about 1 hour, due to a DNS misconfiguration problem [171]. A misconfiguration made Microsoft's public cloud platform, Azure, unavailable for about two and a half hours [140]. Each such incident affected millions of users.

**Hard to debug.** Configuration errors are difficult to diagnose. They usually require great expertise to understand the error root causes. For example, a configuration error in the CentOS kernel prevented a user from mounting a newly-created file system [222]. The user needed deep understanding about the exhibited symptom, and had to re-install kernel modules and also modify configuration option values in several places to get it to work. Techniques to help escape from "configuration hell" are highly demanded [54].

**Actionable.** Unlike software bugs, which can only be fixed by experienced software developers, fixing a software configuration error is *actionable* for software end-users or system administrators. These users are not the software developers and cannot access (much less understand) the source code, but they can fix a configuration error by simply changing the values of certain configuration options.

### *4.2 Example*

We next describe a real scenario in which we used ConfDiagnoser to solve a configuration problem. We received a "bug report" against the Randoop automated test generation tool [148], from a testing expert who had been using Randoop for quite a while. The "bug report" indicated that Randoop terminated normally but failed to generate tests for the NanoXML [144] program.

Although the reported problem is deterministic and fully reproducible, it is a silent, non-crashing failure and is challenging to diagnose. Differing from a crashing error, Randoop did not exhibit a crashing point, dump a stack trace, output an error message, or indicate suspicious program variables that may have incorrect values. Lacking such information makes many techniques such as dynamic slicing [241], dynamic information flow tracking [14], and failure trace analysis [159] inapplicable. In ad-

```
Suspicious configuration option: maxsize

It affects the behavior of predicate:
"newSequence.size() > GenInputsAbstract.maxsize"
(line 312, class: randoop.ForwardGenerator)

This predicate evaluates to true:
  3.3% of the time in normal runs (3830 observations)
  32.5% of the time in the undesired run (2898 observations)
```

Figure 4.1: The top-ranked configuration option in ConfDiagnoser's error report for the motivating example in Section 4.2.

dition, for this scenario, the person who reported the bug had already minimized the bug report: if any part of the configuration or input is removed, Randoop either crashes or no longer exhibits this error. This further makes search-based fault isolation techniques such as delta debugging [227] ineffective.

In fact, this bug report does not reveal a real bug in the Randoop code. Its root cause is that the user failed to set one configuration option. Despite the simplicity of the solution, to the best of our knowledge, no previous configuration error diagnosis technique [13, 14, 159, 203, 209, 238] can be directly applied.

Our technique (and its tool implementation ConfDiagnoser) can diagnose and correct this problem. We first reproduced the error in a ConfDiagnoser-instrumented Randoop version, then ConfDiagnoser diagnosed the error's root cause by analyzing the recorded execution profile. ConfDiagnoser produced a report (Figure 4.1) in the form of an ordered list of suspicious configuration options that should be inspected. The error report in Figure 4.1 suggests that a configuration option named `maxsize` is the most likely one. The report also provides relevant information to explain why: a program predicate affected by `maxsize` behaves dramatically differently between the recorded undesired execution and the correct executions found in ConfDiagnoser's database.

Figure 4.2 shows the relevant code snippet in Randoop. When Randoop generates a new test (line 100, in the form of a method-call sequence), Randoop compares its length with `maxsize` (default value: 100). If the generated sequence's length exceeds this pre-defined limit, Randoop discards it to avoid length explosion in further test generation. Although `maxsize`'s default value was carefully chosen by the Randoop developers and works well for many programs (including those used to test Randoop during its development), the generated sequences for NanoXML are much longer than usual and using `maxsize`'s default value results in 32.5% of the generated sequences being discarded

In class: randoop.main.GenInputsAbstract

```
     //The maxsize configuration option. Default value: 100.
157. public static int maxsize = readFromCommandLine();
```

In class: randoop.ForwardGenerator

```
99.  public ExecutableSequence step() {
100.   ExecutableSequence eSeq = createNewUniqueSequence();
101.   AbstractGenerator.currSeq = eSeq.sequence;
102.   eSeq.execute(executionVisitor);
103.   processSequence(eSeq);
104.   if (eSeq.sequence.hasActiveFlags()) {
105.     componentManager.addGeneratedSequence(eSeq.sequence);
106.   }
107.   return eSeq;
108. }

310. private ExecutableSequence createNewUniqueSequence() {
311.   Sequence newSequence = ...; //create a sequence
312.   if (newSequence.size() > GenInputsAbstract.maxsize) {
313.     return null;
314.   }
315.   if (this.allSequences.contains(newSequence)) {
316.     return null;
317.   }
318.   return new ExecutableSequence(newSequence);
319. }
```

Figure 4.2: Simplified code excerpt from Randoop [148] corresponding to the configuration problem reported in Figure 4.1.

(including sequences that the user wishes to retain). ConfDiagnoser captures such abnormal behavior from Randoop's silent failure, pinpoints the `maxsize` option, and suggests to the user changing its value. The problem is resolved if the user changes `maxsize` to a larger value, for example 1000.

### *4.3 Technique*

Correcting a configuration error can be divided into three separate tasks: reproducing the error, identifying which specific configuration option is responsible for the unexpected behavior, and determining a better value for the configuration option. ConfDiagnoser addresses the second task: finding the root cause of a configuration error.

In ConfDiagnoser, we model a configuration as a set of key-value pairs, where the keys are strings and the values have arbitrary type. This abstraction is offered by the POSIX system environment, the Java Properties API, and the Windows Registry.

### *4.3.1 Overview*

ConfDiagnoser is designed to be used by system administrators and end-users when they encounter an error that they do not know how to fix. It uses three steps, as illustrated in Figure 4.3, to link the undesired behavior to specific root cause configuration options:

- **Configuration Propagation Analysis**. For each configuration option, ConfDiagnoser uses a lightweight dependence analysis, called thin slicing [182], to statically identify the predicates it affects in the source code.

- **Configuration Behavior Profiling**. ConfDiagnoser selectively instruments the program-to-diagnose so that it records the run-time behaviors of affected predicates in an execution profile. When the user encounters a suspected configuration error, the user reproduces the error using the instrumented version of the program.

- **Configuration Deviation Analysis**. ConfDiagnoser selects, from a pre-built database, correct execution profiles that are as similar as possible to the undesired one. Then, it identifies the predicates whose dynamic behaviors deviate the most between correct and undesired executions. The behavioral differences in the recorded predicates provide evidence for what predicates in a program might be behaving abnormally and why. For each deviated predicate, ConfDiagnoser further identifies its affecting configuration options as the likely root causes. Finally, it outputs a ranked list of suspicious configuration options and explanations.

An important component in ConfDiagnoser is the pre-built database, which contains profiles from known correct executions. We envision that the software developers build this database at release time. The database can be further enriched by software users as more correct executions are accumulated. In our experiments (Section 4.5), we built a database of 6–16 execution profiles by running examples from software user manuals, FAQs, discussion mailing list, forum posts, and published papers. We found that even such a small database worked remarkably well for error diagnosis.

Compared to previous approaches [14, 159, 203, 209, 227, 241], ConfDiagnoser has several notable features:

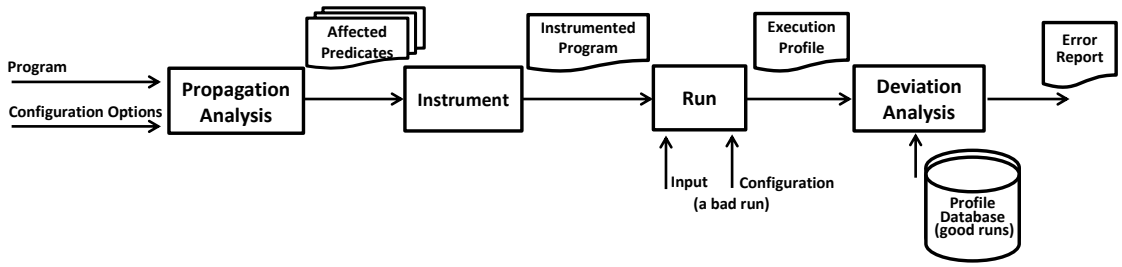



Figure 4.3: The workflow of our configuration error diagnosis technique. "Propagation Analysis" is described in Section 4.3.2. The "Instrument" and "Run" components correspond to the Configuration Behavior Profiling step in Section 4.3.3. "Deviation Analysis" is described in Section 4.3.4.

- **It is fully automated**. ConfDiagnoser does not require a user to specify *when*, *why*, or *how* the program fails. This is different than many well-known automated debugging techniques such as delta debugging [227], information flow analysis [14], and dynamic slicing [241]. ConfDiagnoser also provides an *explanation* of why a configuration option is suspicious.

- **It can diagnose both non-crashing and crashing errors**. Most previous techniques [14, 159, 184, 209] focus exclusively on configuration errors that cause a crash, an error message, or a stack trace. By contrast, ConfDiagnoser diagnoses configuration problems that manifest themselves as either visible or silent failures.

- **It requires no OS-level support.** ConfDiagnoser requires no alterations to the JVM or standard library. This distinguishes our work from competing techniques such as OS-level configuration error troubleshooting [184, 209].

### *4.3.2 Configuration Propagation Analysis*

For each configuration option, Configuration Propagation Analysis statically determines its affected *predicates*. In our context, a *predicate* is a Boolean expression in a conditional or loop statement, whose evaluation result determines whether to execute the following statement or not. A predicate's run-time outcome affects the program control flow. ConfDiagnoser focuses on identifying and monitoring configuration option-affected control flow rather than the values, for two reasons. First, control flow often propagates the majority of configuration-related effects and determines a

program's execution path, while the value of a specific expression may be largely input-dependent. Second, it simplifies reporting because the outcome of a program predicate can only be either true or false. Nevertheless, a program predicate is not the only abstraction our technique can use. Our experiments (Section 4.5) empirically demonstrate that choosing other abstractions, such as monitoring statement-level coverage or method-level invariants, yields less accurate results.

To identify the predicates affected by a configuration option, a straightforward way is to use program slicing [82] to compute a forward slice from the initialization statement of a configuration option. Unfortunately, traditional full slicing [82] is impractical because it includes too much of the program. This is due to conservatism (for example, in handling pointers) and to following both data and control dependences. Figure 4.2 illustrates this problem. Traditional slicing concludes that the predicates in lines 104, 312, and 315 are affected by the configuration option `maxsize`. However, the predicates in lines 104 and 315, though possibly affected by `maxsize`, are actually irrelevant to `maxsize`'s value. That is, the value of `maxsize` controls the length of a generated sequence rather than deciding whether a sequence has an active flag (line 104) or a sequence has been executed before (line 315).

To address this limitation, our technique uses thin slicing [182], which includes only statements that are *directly* affected by a configuration option. Different from traditional slicing [82], thin slicing focuses on data flow from the seed (here, a seed is the initialization statement of a configuration option), ignoring control flow dependencies as well as uses of base pointers. Thin slicing is attractive since it separates pointer computations from the flow of configuration option values and naturally connects a configuration option with its directly affected statements. For example, in the code excerpt of Figure 4.2, a forward thin slice computed for `maxsize` includes only the predicate in line 312. Section 4.5 empirically demonstrates that thin slicing is a better choice than traditional full slicing for our purposes.

### *4.3.3 Configuration Behavior Profiling*

This step instruments the tested program offline by inserting code to record how often each predicate evaluates to true at run time.

Executing the instrumented program produces an *execution profile*, which consists of a set of *predicate profiles*. Each predicate profile is a 4-tuple consisting of a configuration option, one of

its affected predicates, the predicate's execution count, and how many times it evaluated to true. For example, suppose the predicate on line 312 has been executed 100 times, of which 30 times it evaluated to true. ConfDiagnoser creates the following predicate profile:
⟨`maxsize`, `newSequence.size() > maxsize`, 100, 30⟩.

Such predicate profiles are by no means complete in recording the whole execution. However, they capture sufficient information to reason about the correlative effects of configurations and how a configuration option relates to software's behavior, as shown by our experiments (Section 4.5). Collecting these profiles imposes only moderate performance impact.

### *4.3.4 Configuration Deviation Analysis*

ConfDiagnoser starts error diagnosis after obtaining the execution profile from an undesired execution. It selects similar profiles from known correct executions, compares each selected profile with the undesired one to identify the most behaviorally-deviated predicates, and then determines the likely root cause options.

#### *Selecting Similar Execution Profiles for Comparison*

ConfDiagnoser's database contains profiles from known correct executions. These execution profiles can be dramatically different from another. To avoid reporting irrelevant differences when determining how and why the observed execution profile behaves differently from the correct ones, ConfDiagnoser first compares the undesired profile with the correct profiles, then selects a set of similar ones as the basis of diagnosis.

ConfDiagnoser first converts each execution profile $e$ into a $n$-dimensional vector $v_e = \langle r_{e1}, r_{e2}, ..., r_{en} \rangle$, where $n$ is the number of predicates affected by configuration options and each $r_{ei}$ is a ratio representing how often the $i$-th predicate profile evaluated to true at run time. If a predicate has never been executed in an execution, ConfDiagnoser uses 0 as its ratio.

ConfDiagnoser computes the similarity of two execution profiles $e$ and $f$ by computing the cosine similarity from information retrieval [210] of $v_e$ and $v_f$.

$$Similarity(e, f) = cos\_sim(v_e, v_f) = \frac{\sum_{i=1}^{n} r_{ei} \times r_{fi}}{\sqrt{\sum_{i=1}^{n} r_{ei}^2} \times \sqrt{\sum_{i=1}^{n} r_{fi}^2}}$$

This similarity metric compares two execution profiles based on control flow taken (approximated by how often each predicate evaluated to true). Its value ranges from 0 meaning completely different predicate behavior, to 1 meaning the same predicate behavior, and in-between values indicating intermediate similarity.

A crashing error sometimes happens soon after the program is launched, so the resulting execution profile is much smaller than most correct execution profiles. To avoid comparing un-executed predicates, when diagnosing a crashing error, ConfDiagnoser reduces each correct execution profile by only retaining the predicates executed by the crashing profile, and then uses the reduced profile for comparison.

Given an undesired execution profile, ConfDiagnoser selects all execution profiles (or the reduced profiles for a crashing error) from the database with a $Similarity$ value above a threshold (default value: 0.9, as used in our experiments).

*Identifying Deviated Predicates*

Our automated error diagnosis approach compares an undesired execution profile with a set of *similar* and *correct* execution profiles. The behavioral differences in the recorded predicates provide evidence for what parts of a program might be incorrect and why.

ConfDiagnoser characterizes the dynamic behavior of a predicate by how often it was evaluated (i.e., the number of observed executions), and how often it evaluated to true (i.e., the true ratio). The true ratio is more important, but it is less dependable the fewer times the predicate has been evaluated.

We define the following $\phi$ metric, which combines sensitivity (informally, the need for multiple observations) and specificity (informally, the true ratio) in a standard way by computing their harmonic mean.

$$\phi(e,p) = \frac{2}{1/trueRatio(e,p) + 1/totalExecNum(e,p)}$$

In $\phi(e,p)$, $trueRatio(e,p)$ is the ratio of executions of the predicate $p$ that evaluated to true in $e$, and $totalExecNum(e,p)$ is the the total number of executions of predicate $p$ in $e$. To smooth corner cases, if a predicate $p$ is not executed in $e$, i.e., $totalExecNum(e,p) = 0$, then $\phi(e,p)$ returns 0; and if a predicate $p$'s true ratio is 0, i.e., $trueRatio(e,p) = 0$, then $\phi(e,p)$ returns

$1/totalExecNum(e, p)$.

The following $Deviation$ metric compares a predicate $p$ across two execution profiles $e$ and $f$. A larger $Deviation$ value indicates that the behavior is more different.

$$Deviation(p, e, f) = |\phi(e, p) - \phi(f, p)|$$

Often $1/trueRatio(e, p) \gg 1/totalExecNum(e, p)$, and then the value of $Deviation(p, e, f)$ depends primarily on $p$'s true ratio difference between execution profiles $e$ and $f$.

ConfDiagnoser computes the $Deviation$ value for each predicate $p$ appearing in two execution profiles $e$ and $f$, and ranks them in decreasing order. The two execution profiles $e$ and $f$ are for the same program, so they have exactly the same set of predicates affected by configuration options.

*Linking Predicates to Root Causes*

ConfDiagnoser links the behaviorally-deviated predicates to their root cause configuration options by using the results of thin slicing (computed by the Configuration Propagation Analysis step in Section 4.3.2). ConfDiagnoser identifies the affecting configuration options for each deviated predicate, and treats the configuration option affecting a higher-ranked deviated predicate as the more likely root cause. If a predicate is affected by multiple configuration options, ConfDiagnoser prefers options whose initialization statements are *closer* to the deviated predicate (in terms of breath-first search distance in the dependence graph of thin slicing). This heuristic is based on the intuition that statements closer to the predicate seem more likely to be relevant to its behavior.

When multiple correct execution profiles are selected for comparison, ConfDiagnoser first produces a ranked list of root cause configuration options for each comparison pair, and then outputs a final list by using majority voting over all ranking lists. In the final ranking list, one configuration option ranks higher than another if it ranks higher in more than half of the ranking lists. Our implementation breaks possible cycles by arbitrarily ranking the involved options, but this did not occur in our experiments.

In the final output report (e.g., Figure 4.1), ConfDiagnoser generates a brief explanation for each behaviorally-deviated predicate by showing the difference between the predicate's true ratio during correct executions from the database and during the undesired execution.

### *4.3.5 Discussion*

ConfDiagnoser focuses specifically on configuration errors, assuming the application code is correct but the software is inappropriately configured so that it does not behave as desired. We next discuss some design issues in ConfDiagnoser.

**Differences between program inputs and configuration options.** We took the list of configuration options for each subject program from its manual. Typically, the manual calls an input a configuration option when it controls a program's control flow rather than producing result data. A configuration option is often supplied via a command-line flag or configuration file.

**Why not use profiles from unit test executions?** ConfDiagnoser's database stores correct profiles from complete executions that start at the main method. ConfDiagnoser does not use profiles from unit test executions, which check the correctness of a single program component and produce an incomplete execution profile that is not representative of the whole program workflow.

**Why not store profiles from failing executions in the database?** We envision the profile database is built by developers at release time. It is more natural and easier for a developer to provide correct execution profiles, instead of anticipating and enumerating the possible errors a user may encounter.

**What if a similar execution profile is not available?** ConfDiagnoser's effectiveness largely depends on the availability of similar execution profiles from the database. For a given undesired execution profile, lacking a similar profile in ConfDiagnoser's database may lead ConfDiagnoser to produce less useful results. It also indicates inadequacy of the tests from which the database was constructed. Future work should remedy this problem. One possible approach is to synthesize a new execution, either by generating a new input for the program [236] or by mutating an existing execution [185].

## *4.4 Tool Implementation*

We implemented a tool, called ConfDiagnoser, on top of the WALA framework [202]. Our tool analyzes Java bytecode. It statically computes the affected predicates for each configuration option, assigns a unique ID for each affected predicate, and then performs offline instrumentation. The runtime behavior of all affected predicates is recorded in a file.

For a Java program, ConfDiagnoser does not analyze the standard JDK library and all the dependent libraries. We believe this approximation is reasonable, since a configuration option set in

client software usually does not affect the behaviors of its dependent libraries.

## *4.5 Evaluation*

Our evaluation answers the following research questions:

- How effective is ConfDiagnoser in error diagnosis? ConfDiagnoser's effectiveness can be reflected by:
  - the absolute ranking of the actual root cause in ConfDiagnoser's output
  - the time cost of error diagnosis
  - comparison with a previous configuration error diagnosis technique
  - comparison with two fault localization techniques
- What are the effects of using full slicing [82] rather than thin slicing [182] to identify the affected predicates? What are the effects of varying comparison execution profiles? These are two internal design choices.

### *4.5.1 Subject Programs*

We evaluated ConfDiagnoser on 5 Java programs shown in Figure 4.4. Randoop [148] is an automated test generator for Java programs. Weka [207] is a toolkit that implements machine learning algorithms. Our evaluation uses only its decision tree module. JChord [93] is a program analysis platform that enables users to design, implement, and evaluate static and dynamic program analyses for Java. Synoptic [187] mines a finite state machine model representation of a system from logs. Soot [179] is a Java optimization framework for analyzing and transforming Java bytecode.

*Configuration Errors*

We collected 14 configuration errors, listed in Figure 4.5. We have evaluated all the configuration errors we found; we did not select only errors on which ConfDiagnoser works well. The misconfigured values include enumerated types, numerical ranges, regular expressions, and strings. The 5 non-crashing errors are collected from actual bug reports, mailing list posts, and our own experience.

| Program (version) | LOC | #Config Options | #Profiles |
|---|---|---|---|
| Randoop (1.3.2) | 18587 | 57 | 12 |
| Weka Decision Trees (3.6.7) | 3810 | 14 | 12 |
| JChord (2.1) | 23391 | 79 | 6 |
| Synoptic (trunk, 04/17/2012) | 19153 | 37 | 6 |
| Soot (2.5.0) | 159273 | 49 | 16 |

Figure 4.4: Subject programs. Column "LOC" is the number of lines of code, as counted by LOCC [127]. Column "#Config Options" is the number of configuration options. Column "#Profiles" is the number of execution profiles in the pre-built database.

| Error ID | Program | Description |
|---|---|---|
| Non-crashing errors | | |
| 1 | Randoop | No tests generated |
| 2 | Weka | Low accuracy of the decision tree |
| 3 | JChord | No datarace reported for a racy program |
| 4 | Synoptic | Generate an incorrect model |
| 5 | Soot | Source code line number is missing |
| Crashing errors | | |
| 6 | JChord | No main class is specified |
| 7 | JChord | No main method in the specified class |
| 8 | JChord | Running a nonexistent analysis |
| 9 | JChord | Invalid context-sensitive analysis name |
| 10 | JChord | Printing nonexistent relations |
| 11 | JChord | Disassembling nonexistent classes |
| 12 | JChord | Invalid scope kind |
| 13 | JChord | Invalid reflection kind |
| 14 | JChord | Wrong classpath |

Figure 4.5: The 14 configuration errors used in evaluating ConfDiagnoser.

The 9 crashing errors, taken from [159], were previously used to evaluate the ConfAnalyzer tool. All 14 configuration errors are minimal: if any part of the configuration or input is removed, the software either crashes or no longer exhibits the undesired behavior.

### *4.5.2 Evaluation Procedure*

For each subject program, we constructed a profile database by running existing (correct) examples from its user manual, FAQs, discussion mailing list, forum posts, and published papers [148,159]. We spent 3 hours per program, on average, and obtained 6–16 execution profiles. The average size of the profile database is 35MB, and the largest one (Randoop's database) is 72MB.

We made a simple syntactic change to JChord, which affected 24 lines of code. This change does not modify JChord's semantics; rather, it just encapsulates scattered configuration option initialization statements as static class fields, which simplifies specifying the seed statement in performing slicing.

When diagnosing a configuration error, we first reproduced the error on a ConfDiagnoser-instrumented program to obtain the execution profile. Then, we ran ConfDiagnoser on the obtained execution profile to identify its root causes.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

### *4.5.3 Results*

Figure 4.6 shows the experimental results.

*Accuracy in Diagnosing Configuration Errors*

As shown in Figure 4.6, ConfDiagnoser is highly effective in pinpointing the root cause of misconfigurations. For all 5 non-crashing errors and 5 of the 9 crashing errors, it lists the actual root cause as one of the top 3 options.

**Non-crashing configuration errors.** ConfDiagnoser is particularly effective in diagnosing non-crashing configuration errors, which are not supported by most other tools. The average rank of the root cause in ConfDiagnoser's output is 1.6. The primary reason is ConfDiagnoser's ability to identify the behaviorally-deviated predicates through execution profile comparison. The top-ranked deviated predicates often provide useful clues about what parts of a program might be abnormal and why.

We use the non-crashing error in Weka as an example to illustrate this point. Weka's decision tree implementation is highly tuned, achieving 70–90% accuracy on its included examples. However, its accuracy drops to 62% on a different dataset we experimented on. We used ConfDiagnoser to diagnose this problem by first building a database by running Weka on its examples, and then obtaining the undesired execution profile by running it on our dataset. As a result, ConfDiagnoser outputs the following report (only the top option is shown):

```
Suspicious configuration option: m_numFolds
```

| Error ID. Program | Root Cause Configuration Option | #Options | Program Output | ConfDiagnoser | | ConfAnalyzer | Coverage Analysis | Invariant Analysis | ConfDiagnoser w/ Full Slicing |
|---|---|---|---|---|---|---|---|---|---|
| | | | Rank | #Profiles | Rank | Rank | Rank | Rank | Rank |
| Non-crashing errors | | | | | | | | | |
| 1. Randoop | `maxsize` | 57 | **N** | 10 / 12 | 1 | **X** | 13 | **N** | 46 |
| 2. Weka | `m_numFolds` | 14 | **N** | 2 / 12 | 1 | **X** | 4 | 5 | 9 |
| 3. JChord | `chord.datarace.eqth` | 79 | **N** | 2 / 6 | 3 | **X** | 38 | 2 | 73 |
| 4. Synoptic | `partitionRegExp` | 37 | **N** | 2 / 6 | 1 | **X** | 1 | **N** | 6 |
| 5. Soot | `keep_line_number` | 49 | **N** | 6 / 16 | 2 | **X** | 46 | **N** | **N** |
| Average | | 47.2 | 23.6 | 3.6 / 10.4 | 1.6 | 23.6 | 20.4 | 15.7 | 31.7 |
| Crashing errors | | | | | | | | | |
| 6. JChord | `chord.main.class` | 79 | 1 | 4 / 6 | 1 | 1 | 1 | 4 | 5 |
| 7. JChord | `chord.main.class` | 79 | 1 | 5 / 6 | 1 | 1 | 1 | 4 | 5 |
| 8. JChord | `chord.run.analyses` | 79 | 1 | 5 / 6 | 17 | 1 | 17 | 22 | 21 |
| 9. JChord | `chord.ctxt.kind` | 79 | 1 | 3 / 6 | 1 | 3 | 25 | 30 | 75 |
| 10. JChord | `chord.print.rels` | 79 | 1 | 2 / 6 | 15 | 1 | 20 | 25 | 24 |
| 11. JChord | `chord.print.classes` | 79 | 1 | 4 / 6 | 16 | 1 | 13 | 17 | 22 |
| 12. JChord | `chord.scope.kind` | 79 | 1 | 5 / 6 | 1 | 1 | 1 | **N** | 10 |
| 13. JChord | `chord.reflect.kind` | 79 | 1 | 6 / 6 | 1 | 3 | 5 | 9 | 11 |
| 14. JChord | `chord.class.path` | 79 | 1 | 4 / 6 | 8 | **N** | 21 | 26 | 6 |
| Average | | 79 | 1 | 4.2 / 6 | 6.7 | 5.7 | 11.5 | 19.5 | 19.8 |

Figure 4.6: Experimental results in diagnosing software configuration errors. Column "Root Cause Configuration Option" shows the actual root cause configuration option. Column "#Options" shows the number of configuration options, taken from Figure 4.4. Column "Program Output" shows the rank of the root cause as indicated by the program's output, such as an error message. Column "ConfDiagnoser" shows the results of using our technique. Column "#Profiles" shows the number of similar execution profiles selected from the pre-built database for comparison, and the number of executions in the database. For each technique, Column "Rank" shows the absolute rank of the actual root cause in its output (lower is better). "**X**" means the technique is not applicable (i.e., requiring a crashing point), and "**N**" means the technique does not identify the actual root cause. When computing the average rank, each "**X**" or "**N**" is treated as half of the number of configuration options, because a user would need to examine on average half of the options to find the root cause. Column "ConfAnalyzer" shows the results of using a previous technique [159]; the data in this column is taken from [159]. Columns "Coverage Analysis" and "Invariant Analysis" show the results of using two fault localization techniques as described in Section 4.5.3. Column "ConfDiagnoser w/ Full Slicing" shows the results of using full slicing [82] to compute the affected predicates (Section 4.5.3).

```
It affects the behavior of predicate:
"numFold < numInstances() % numFolds"
(line 1354, class: weka.core.Instances)

This predicate evaluates to true:
  20% of the time in normal runs (4 observations)
  70% of the time in the undesired run (10 observations)
```

The above report reveals an important fact about the low accuracy. The predicate `numFold < numInstances() % numFolds` controls the depth of a decision tree. Its true ratio is substantially higher in the undesired execution than in normal executions. A higher true ratio leads to a deeper tree that is more likely to overfit the training data and yield low accuracy on the testing data. To resolve this problem, we changed `m_numFolds` value from 2 to 3 to reduce the tree depth, and gained a 5% performance increase.

**Crashing configuration errors.** Crashing errors are often easy for a user to diagnose. This is because a crashing error often produces a stack trace or error message with valuable diagnosis clues. In fact, for the crashing errors selected by the ConfAnalyzer authors, the user is always led to the root cause by the program output, without the need for further analysis. For error #6, JChord throws a `NoClassDefFoundError` when loading a user-specified class. This error reminds the user that a non-existent class might be provided. For error #7, JChord outputs an error message of "Could not find main class [...] or main method in that class" that explicitly informs the user that the configuration option to specify a main class might be wrong. For errors #8–13, JChord outputs the relevant configuration option in its error message. For error #14, JChord throws a `ClassNotFoundException` (for the main class) that reminds the user to check the classpath setting.

ConfDiagnoser is more effective in diagnosing non-crashing errors (average rank: 1.6) than crashing errors (average rank: 6.7). Most of the crashing errors occur soon after the program is launched, so ConfDiagnoser lacks enough predicate behavior observations. Many predicates are executed only once, so their $Deviation$ scores (Section 4.3.4) turn out to be the same; and the BFS-distance-based heuristic to resolve ties (Section 4.3.4) works only half the time.

*Performance of ConfDiagnoser*

We measured ConfDiagnoser's performance in two ways: the time cost in diagnosing an error and the overhead introduced in reproducing an error in a ConfDiagnoser-instrumented program.

As shown in Figure 4.7, the performance of ConfDiagnoser is reasonable. On average, it uses less than 4 minutes to diagnose each configuration error (including the time to compute thin slices and

| Error ID. Program | Run-time Slowdown (×) | ConfDiagnoser time (seconds) | |
|---|---|---|---|
| | | Thin Slicing | Error Diagnosis |
| Non-crashing errors | | | |
| 1. Randoop | 1.1 | 50 | $< 1$ |
| 2. Weka | 1.2 | 43 | $< 1$ |
| 3. JChord | 13.2 | 147 | 82 |
| 4. Synoptic | 3.6 | 24 | $< 1$ |
| 5. Soot | 3.1 | 95 | 21 |
| Mean | 2.9 | 72 | 21 |
| Crashing errors | | | |
| 6. JChord | 2.4 | 147 | 79 |
| 7. JChord | 1.4 | 147 | 75 |
| 8. JChord | 1.5 | 147 | 17 |
| 9. JChord | 28.5 | 147 | 30 |
| 10. JChord | 13.7 | 147 | 13 |
| 11. JChord | 65.1 | 147 | 10 |
| 12. JChord | 1.6 | 147 | 83 |
| 13. JChord | 1.9 | 147 | 8 |
| 14. JChord | 1.4 | 147 | 80 |
| Mean | 4.3 | 147 | 44 |

Figure 4.7: ConfDiagnoser's performance. The run-time slowdown column shows the cost of reproducing the error in an instrumented version of the subject program, and the mean is the geometric mean. The ConfDiagnoser time has been divided into two parts — computing thin slices and diagnosing an error — and the mean is the arithmetic mean.

the time to recommend suspicious configuration options). Computing thin slices for all configuration options is expensive. However, this step is one-time effort per program and the computed slices can be cached to share across diagnoses.

The performance overhead to reproduce the buggy behavior varies among applications. The current tool implementation imposes a substantial slowdown when reproducing errors 3, 9, 10, and 11 in a ConfDiagnoser-instrumented version. This is due to ConfDiagnoser's naive, inefficient instrumentation code, which we have made no effort to optimize. Even so, an error can be reproduced in less than 4 minutes on average, with a worst case of 13 minutes.

*Comparison with a Previous Technique*

We compared ConfDiagnoser with ConfAnalyzer, a dynamic information flow-based technique [159]. We chose ConfAnalyzer because it is the most recent technique and also one of

the most precise configuration error diagnosis techniques in the literature. ConfAnalyzer tracks the flow of labeled objects through the program dynamically, and treats a configuration option as a root cause if its value may flow to a crashing point. ConfAnalyzer works well for most of the crashing errors (all of which are from the ConfAnalyzer paper [159]), though as described above these are easy to diagnose even without tool support. However, ConfAnalyzer cannot diagnose non-crashing errors.

The experimental results of ConfAnalyzer are shown in Figure 4.6 (column "ConfAnalyzer"). For the 9 crashing errors, ConfDiagnoser produced better results for 3 of them, the same results for another 3, and worse results for the remaining 3.

ConfAnalyzer performs best on the easiest crashing errors: those having short execution paths, when an error exhibits almost immediately after the software is launched. In such cases, only a small number of configuration options are initialized and few of them can flow to the crashing point. ConfDiagnoser fails to produce a good diagnosis for these errors, because it cannot identify the statistically-behaviorally-deviated predicates based on the limited observation of program behaviors.

ConfAnalyzer outputs less accurate or no results for errors where the root cause option value flows into containers or system calls (e.g., error #14 in Figure 4.6). ConfDiagnoser can reason (to some extent) about the *consequence* of such a misconfiguration based on the observed predicate behaviors.

*Comparison with Two Fault Localization Techniques*

Another possible way to diagnose a configuration error is to leverage existing fault localization techniques, by treating the undesired execution as a failing run and all correct executions (in the database) as passing runs. We next compare ConfDiagnoser with two state-of-the-art techniques:

- **Statement-level Coverage Analysis**. This technique treats statements covered by the undesired execution profile as potentially buggy, and statements covered by the correct execution profiles as correct. Then, it leverages a well-known fault localization technique, Tarantula [102], to rank the likelihood of each statement being buggy, and queries the results of thin slicing to identify its affecting configuration options as the root causes. The results are essentially the same for a variant of coverage analysis: using thin slicing to compute all affected statements,

and only monitoring the coverage of such affected statements.

- **Method-level Invariant Analysis**. This technique stores invariants detected by Daikon [49] from correct executions in the database. It treats a method as having suspicious behavior if its observed invariants from the undesired execution are different from the invariants stored in the database [132]. This technique ranks a method's suspiciousness by the number of different invariants, and queries the results of thin slicing to identify its affecting configuration options as the root causes.

In Coverage Analysis, the statement-level granularity is *too fine-grained*. Many statements have exactly the same coverage in the failing/passing executions, and thus have the same suspiciousness score as computed by Tarantula [102]. Furthermore, Tarantula only records whether a statement has been executed or not but does not record how a statement is executed (e.g., how often a predicate evaluates to true). The combination of these two factors causes the low accuracy.

In Invariant Analysis, the method-level granularity is *too coarse-grained*. Invariant detection techniques like Daikon [49] only check program states at method entries and exits to infer likely pre- and post-conditions, and thus are less sensitive to control flow details within a method (e.g., a predicate's true ratio). In our study, Invariant Analysis failed to diagnose 3 errors. For Synoptic, it failed to infer invariants. For Soot and Randoop, it reported the same invariants over undesired and correct executions, for the method containing behaviorally-deviated predicates.

This experiment suggests that we cannot treat configuration options as just another regular program input, and then directly apply existing fault localization techniques [102, 132] to find the error causes. The primary reason is that, unlike a program input, a configuration option is often used to control a program's control rather than produce result data. Thus, focusing on the behaviors of relevant predicates as our tool does may be a good choice.

*Evaluation of Two Design Choices in ConfDiagnoser*

We investigate the effects of:

- using traditional full slicing [82] rather than thin slicing [182] in the Configuration Propagation Analysis step (Section 4.3.2) to compute the affected predicates. Figure 4.6 (Column "Full

| Error ID. Program | Rank of the Actual Root Cause | | |
|---|---|---|---|
| | All Profiles | Random Selection | Similarity-Based |
| Non-crashing errors | | | |
| 1. Randoop | 1 | 2 | 1 |
| 2. Weka | 7 | 6 | 1 |
| 3. JChord | 16 | 19 | 3 |
| 4. Synoptic | 1 | 1 | 1 |
| 5. Soot | 13 | 13 | 2 |
| Average | 7.6 | 8.2 | 1.6 |
| Crashing errors | | | |
| 6. JChord | 1 | 1 | 1 |
| 7. JChord | 1 | 1 | 1 |
| 8. JChord | 17 | 17 | 17 |
| 9. JChord | 1 | 1 | 1 |
| 10. JChord | 15 | 15 | 15 |
| 11. JChord | 16 | 16 | 16 |
| 12. JChord | 25 | 25 | 1 |
| 13. JChord | 1 | 1 | 1 |
| 14. JChord | 9 | 9 | 8 |
| Average | 9.4 | 9.4 | 6.7 |

Figure 4.8: Comparison with different execution profile selection strategies (Section 4.5.3). The last column "Similarity-based" is the selection strategy used in ConfDiagnoser, and the data in that column is taken from Figure 4.6.

Slicing") shows the results.

- varying the comparison execution profiles from the pre-built database. In particular, we compare the similarity-based selection strategy used in ConfDiagnoser (Section 4.3.4) with two alternatives: selecting all available profiles in the database, and randomly selecting the same number of profiles as ConfDiagnoser uses from the database. Figure 4.8 shows the results. For random selection, we performed the experiment 10 times and report the average.

As shown in Figure 4.6 (Column "Full Slicing"), ConfDiagnoser achieves substantially less accurate results when using full slicing. The primary reason is that full slicing includes too many irrelevant statements that are only *indirectly* affected by a configuration option value but not pertinent to the task of error diagnosis. In many cases, monitoring the control flow of such indirectly-affected predicates and then linking their behaviors to configuration options leads to low accuracy. Furthermore, performing full slicing is much more expensive than thin slicing; in our experiments,

the full slicing algorithm ran out of memory on Soot.

As shown in Figure 4.8, varying the selection strategy for correct traces can affect the results, depending on the application being analyzed. Using all available execution profiles or randomly selecting execution profiles is less effective, because they make ConfDiagnoser report many irrelevant differences between an undesired execution and a dramatically different execution. When diagnosing a crashing error, ConfDiagnoser is less sensitive to the comparison execution profiles. This is because crashing profiles are often much smaller, executing fewer predicates before reaching the crashing points; and ConfDiagnoser reduces each correct execution profile before diagnosis (Section 4.3.4). Thus, many irrelevant differences have already been removed.

Because the trace selection strategy improved the accuracy of ConfDiagnoser, we tried applying it to Coverage Analysis and Invariant Analysis as well. That is, we supplied those analyses not with the full database of good runs, but with the runs most similar to the bad run. This approach degraded the accuracy of the other tools beyond the results shown in Figure 4.6, even though it helped ConfDiagnoser. The reason is that the suspiciousness of a statement or method is inversely proportional to the number of correct execution profiles that cover it. When using fewer correct execution profiles, more statements or methods have the same suspiciousness scores.

### *4.5.4 Discussion*

***Limitations.*** The experiments indicate several limitations of our technique. First, we only focus on named configuration options with a common key-value semantics, and our implementation and experiments are restricted to Java. Second, we evaluated ConfDiagnoser on configuration errors involving just one mis-configured option. Third, our implementation currently does not support debugging non-deterministic errors. For non-deterministic errors, ConfDiagnoser could potentially leverage a deterministic replay system that can capture an undesired non-deterministic execution and faithfully reproduce it for later analysis. Fourth, ConfDiagnoser's effectiveness largely depends on the availability of a similar but correct execution profile. Using an arbitrary execution profile (as we demonstrated in our experiments by using random selection) may significantly affect the results.

***Threats to Validity.*** There are three major threats to validity in our evaluation. First, the 5 programs and the configuration errors may not be representative. Thus, we cannot claim the results can be generalized to an arbitrary program. For example, we did not evaluate ConfDiagnoser to diagnose

misconfigurations that cause poor performance. Second, ConfDiagnoser focuses specifically on configuration errors, assuming the application code is correct. Furthermore, in our experiments, all 14 errors have been minimized (as end-users often do when reporting an error). ConfDiagnoser might produce different error diagnosis results on buggy application code with non-minimized inputs. A user who did not know whether a program was misbehaving due to a bug in the code or an incorrect configuration option would need to apply multiple debugging techniques. We have not yet formulated guidance regarding when the user should give up on ConfDiagnoser and assume the error is not related to a configuration option. Third, we only compared two dependence analyses (thin slicing and full slicing), three abstraction granularities (at the predicate level, statement level [102], and method level [49]), and three other tools (ConfAnalyzer, Coverage Analysis, and Invariant Analysis) in our evaluation. Using other dependence analyses, abstraction levels, or tools might achieve different results.

***Experimental Conclusions.*** We have three chief findings: (1) ConfDiagnoser is effective in diagnosing both crashing and non-crashing configuration errors with a small profile database. (2) ConfDiagnoser produces more accurate diagnosis than approaches leveraging existing fault localization techniques [102, 132]. (3) Thin slicing and selection of similar comparison traces permit ConfDiagnoser to produce a more accurate diagnosis than other approaches.

### *4.6 Summary*

This chapter presented a practical technique (and its tool implementation, called ConfDiagnoser) for diagnosing configuration errors. The key idea of ConfDiagnoser is comparing an undesired execution trace with multiple correct execution traces and using statistical analysis to link the undesired behavioral difference to the root-cause configuration options. Our experimental results show that ConfDiagnoser is effective in diagnosing both crashing and non-crashing configuration errors, and it does so with a small profile database.

# Chapter 5

# DIAGNOSING CONFIGURATION ERRORS IN EVOLVING SOFTWARE SYSTEMS

ConfDiagnoser supports software configuration error diagnosis in a single software version, but it is not cognizant of software evolution. In practice, during software evolution, developers may change how the configuration options behave. When upgrading to a new software version, users may need to re-configure the software by changing the values of certain configuration options.

This chapter presents a technique, called ConfSuggester, to troubleshoot configuration errors caused by software evolution. Section 5.1 introduces the problem of configuration error diagnosis in evolving software, Section 5.2 presents a study to show that configuration changes do arise in practice, Section 5.3 details the ConfSuggester technique, Section 5.4 describes the tool implementation, Section 5.5 shows the experiments, and Section 5.6 concludes.

## *5.1 Problem*

Continual change is a fact of life for software systems. Among software changes, configuration changes are prevalent. In Section 5.2, we studied 8 real-world configurable software systems and found configuration changes in *every* studied version of *each* system.

During software evolution, developers may change how the configuration options behave. In many cases, reusing the old version's configuration can lead the new software version to exhibit *undesired* behaviors, even if the software is working exactly as *designed*. Therefore, when upgrading to a new software version, users may need to re-configure the software by changing the values of certain configuration options

Take the popular JMeter performance testing tool as an example. In version 2.8, the testing report is saved as an XML file after running an example command (`jmeter -n -t ../threadgroup.jmx -l ../output.jtl -j ../test.log`) from the user manual. However, after upgrading to version 2.9, the same command saves the testing report in a CSV file. Further, all JMeter regression

| Program | Versions | Years | LOC (latest version) | Language |
|---|---|---|---|---|
| MySQL | 5.1, 5.5, 5.6, 5.7 | 3 | 1565212 | C/C++ |
| Apache | 2.0, 2.2, 2.4 | 3 | 139178 | C/C++ |
| Firefox | 7.0–22.0 (16 versions) | 3 | 8237915 | C/C++ |
| Randoop | 1.2.1, 1.3.2, 1.3.3 | 6 | 19511 | Java |
| Weka | 3.4, 3.5, 3.6, 3.7 | 2 | 288369 | Java |
| JChord | 1.0, 2.0, 2.1 | 4 | 26617 | Java |
| Synoptic | 0.04, 0.05, 0.1 | 2 | 19153 | Java |
| JMeter | 2.6, 2.7, 2.8, 2.9 | 2 | 91979 | Java |

Figure 5.1: The open-source software systems we studied and their characteristics. Column "Years" is the active development period for the selected versions.

tests pass on the updated version. The new JMeter version behaves as *designed* but *differently* than a user was expecting.

Our technique (and its tool implementation ConfSuggester) aims to diagnose such configuration errors introduced during software evolution. For the JMeter example, a user first demonstrates the different behaviors on two ConfSuggester-instrumented JMeter versions. Then, ConfSuggester analyzes the recorded execution traces produced by the two instrumented versions, and outputs a ranked list of suspicious configuration options that may need to be changed. At the top of the list is the `output_format` option with a default value of `CSV` in version 2.9. To resolve this problem, users only need to change its value to `XML`.

## *5.2 An Empirical Study of Real-World Configuration Changes*

In the software engineering literature, despite a rich body of software change analysis work [39, 43, 118, 189, 223, 244], software configuration changes across multiple versions are less studied. Do configuration changes arise during software evolution in practice? This section describes an initial study of 8 real-world configurable systems to answer this question.

### *5.2.1 Subject Programs and Study Methodology*

Figure 5.1 lists 8 open-source configurable systems used in our study. MySQL [141] is a popular relational database management system. Apache [7] has been the dominant HTTP server on the Internet since 1996. Firefox [55] is an open-source browser available on multiple platforms.

| Program | # Added Options | # Deleted Options | # Modified Options |
|---|---|---|---|
| MySQL | 26 | 24 | 23 |
| Apache | 5 | 0 | 10 |
| Firefox | 28 | 7 | 56 |
| Randoop | 37 | 26 | 2 |
| Weka | 72 | 4 | 13 |
| JChord | 13 | 10 | 5 |
| Synoptic | 3 | 0 | 2 |
| JMeter | 17 | 3 | 12 |
| Total | 201 | 70 | 123 |

Figure 5.2: The total number of new, deleted, and modified configuration options for each subject program.

| **Change Type** | **Description** |
|---|---|
| Bugs | Fix existing bugs |
| Renaming | Change the option name |
| Features | Add, remove, or modify features |
| Reliability | Improve reliability or performance |

Figure 5.3: Types of configuration changes identified in our study from the subject programs in Figure 5.1.

Randoop [160] is an automated test generator for Java programs. Weka [207] is a toolkit that implements machine learning algorithms. JChord [93] is a program analysis platform that enables users to design and implement static and dynamic program analyses for Java. Synoptic [187] mines a finite state machine model representation of a system from logs. JMeter [100] is a tool to load-test functional behavior and measure performance. Each program is highly configurable, and has evolved over a considerable amount of time for the selected versions (2–6 years).

In our study, we manually examined the revision history of each subject program, and searched for 5 keywords (“configuration option”, “add option”, “delete option”, “rename option”, and “change option”) in commit messages and in the change logs. We searched 7022 commit messages and 28 change log entries, in which 422 commit messages and 28 change log entries were matched. For each match, we read the description of the change and the “diff” of the original file to check whether a configuration option is changed. We collected 394 distinct configuration changes in total.

| Program | # Changed Configuration Options | | | |
|---|---|---|---|---|
| | Bugs | Renaming | Features | Reliability |
| MySQL | 0 | 15 | 55 | 3 |
| Apache | 0 | 0 | 11 | 4 |
| Firefox | 28 | 0 | 55 | 8 |
| Randoop | 0 | 2 | 62 | 1 |
| Weka | 1 | 0 | 81 | 8 |
| JChord | 0 | 2 | 24 | 2 |
| Synoptic | 0 | 5 | 0 | 0 |
| JMeter | 7 | 0 | 18 | 7 |
| Total | 36 | 24 | 301 | 33 |

Figure 5.4: The number of configuration changes of each type described in Figure 5.3.

### *5.2.2 Findings*

Figure 5.2 summarizes the identified configuration changes for each subject program. Figure 5.4 further classifies the configuration changes into four categories shown in Figure 5.3 (each change belongs to a single category[1]).

As shown in Figure 5.2, configuration changes occur in the evolution of every subject program. In fact, they occur in every version of each subject program (not shown in Figure 5.2 for brevity).

As shown in Figure 5.4, feature-related configuration changes are the largest group across all subject programs. These changes include adding new configuration options, deleting existing options, or modifying the default value of an option.

Configuration evolution can have unexpected impacts on program behavior. After configuration changes, reusing an old configuration may yield a misconfiguration, causing different results on the new version. Section 5.5 shows concrete examples.

### *5.2.3 Threats to Validity*

Our findings apply in the context of our subject programs and methodology; they might not apply to arbitrary programs.

The configuration changes identified by our methodology are certainly not complete. Our keyword

[1]The "Bugs" change type in Figure 5.3 represents that fixing some bugs led to changes in configurations rather than fixing some buggy configurations.

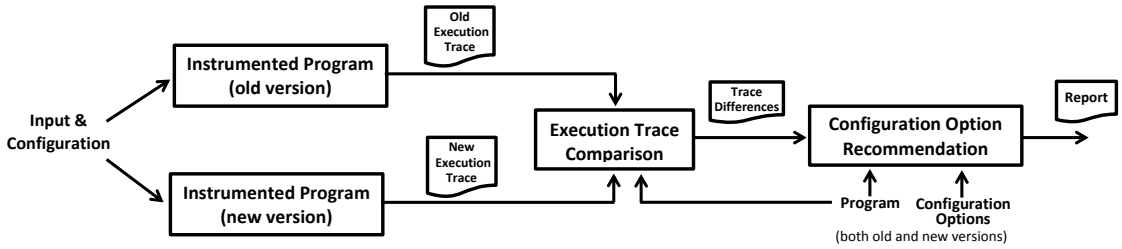



Figure 5.5: The architecture of our ConfSuggester technique. The instrumented program versions and two execution traces are produced by the step in Section 5.3.2. The "Execution Trace Comparison" step is described in Section 5.3.3. The "Configuration Option Recommendation" step is described in Section 5.3.4.

search might have missed some configuration changes. Our methodology only studies changes that are directly made to a software configuration option. We may miss code or environment changes that indirectly affect the software behavior and require users to re-configure the new software version.

## *5.3 Technique*

Like ConfDiagnoser, ConfSuggester models a configuration as a set of key-value pairs, where the keys are strings and the values have arbitrary type, and specifically focuses on software configuration errors. It aims to help two types of users: software end-users who may have problems with software installed on their personal computers, and system administrators who are responsible for maintaining production systems. They can use ConfSuggester to diagnose an unexpected configuration problem during software evolution.

### *5.3.1 Overview*

The key idea of ConfSuggester is to approximate program behavioral differences by control flow differences between two executions (by running the old and new program versions, respectively), and then reason about the control flow differences to identify configuration options that might cause such differences. It uses three steps, as illustrated in Figure 5.5, to link the undesired behavior to specific root-cause configuration options:

- **Instrumentation and Profiling.** ConfSuggester instruments both the old and new program

versions to monitor the execution of each statement as well as the evaluation result of every predicate. Then, it asks the user to demonstrate the different behaviors on the two instrumented program versions.

- **Execution Trace Comparison.** ConfSuggester analyzes the two execution traces to identify the control flow differences. ConfSuggester identifies program predicates that behave differently between the two versions. These behaviorally-deviated predicates and their affected program statements provide evidence about which parts of a program might be behaving abnormally and why.

- **Configuration Option Recommendation.** ConfSuggester uses a lightweight static dependence analysis technique, called thin slicing [182], to attribute control flow differences to specific configuration options. Finally, it outputs a ranked list of suspicious options to the users.

Compared to the ConfDiagnoser technique described in Chapter 4, ConfSuggester differs in two key aspects: (1) ConfDiagnoser diagnoses errors from a single program version, while ConfSuggester is cognizant of software evolution and works on two different versions of the same program. ConfSuggester uses the desired behavior of the old software version as a baseline against which to compare new program behavior, and only reasons about the behavioral differences. (2) ConfSuggester uses a different metric to compare the undesired execution trace with a correct execution trace. Specifically, ConfDiagnoser only uses a predicate's deviation value to reason about the most suspicious options (Section 4.3.4), while ConfSuggester also considers the statements controlled by a predicate's evaluation result (details are described in Section 5.3.3). We empirically compare ConfDiagnoser and ConfSuggester in Section 5.5.3.

### *5.3.2 Instrumentation and Demonstration*

ConfSuggester first instruments both the old and new program versions to monitor the program execution at runtime. ConfSuggester directly instruments the bytecode. The instrumentation consists of two parts:

- For each program predicate (i.e., a branch instruction in bytecode), ConfSuggester inserts one probe before and one probe after it to monitor how frequently the predicate is executed and how often the predicate evaluates to true. In our context, a predicate is a Boolean expression in a conditional or loop statement, whose evaluation result affects the program control flow by determining whether to execute the following statement or not. This instrumentation strategy is also used in ConfDiagnoser.

- For each of the other statements, ConfSuggester inserts one probe before it to monitor whether the statement gets executed or not at runtime. The statement execution information is used to calculate the number of executed statements controlled by a predicate (Section 5.3.4).

After instrumentation, ConfSuggester asks the user to demonstrate the different behaviors on the two instrumented program versions, using the same input and configuration. Demonstration is one of the simplest ways for an end-user to describe her problem; and it is easier than writing specifications or scripts of any form.

Executing the instrumented program produces an execution trace, which consists of a sequence of executed statements as well as the execution count and evaluation result of each predicate. The execution trace captured by ConfSuggester is by no means complete in recording the full program behavior; it only captures the control flows a program is taking. As demonstrated in our experiments, such control flow information serves as a good approximation to diagnose the undesired program behavior.

### *5.3.3 Execution Trace Comparison*

In this step, ConfSuggester compares two execution traces from two program versions and identifies the control flow differences between them. ConfSuggester focuses on the recorded behavior of each predicate. First, it statically matches each predicate in the old source code to its counterpart in the new source code. Then, it identifies all predicates that behave differently across the execution traces.

**Auxiliary functions**:
matches($s$, $s'$): return whether two statements $s$ and $s'$ are matched. Details are explained in Section 5.3.3.
BFS($s$, $cfg$, $d$): return a list of statements reachable from statement $s$ in $cfg$ within $d$ graph edges in Breath-First Search (BFS) order.
firstMatchedPair($stmtList_1$, $stmtList_2$): return the first matched statement pair $\langle s, s' \rangle$ such that $s \in stmtList_1$, $s' \in stmtList_2$, and matches($s$, $s'$) return true. Return `null` if no such pair exists.

**Input**: two methods from two software versions: $m_{old}$ and $m_{new}$,
a maximum lookahead value $lh$. (Our experiment uses $lh = 5$.)
**Output**: matched statements between $m_{old}$ and $m_{new}$.
matchStatements($m_{old}$, $m_{new}$, $lh$)

```
1:  matchedStmts ← new Map⟨Statement, Statement⟩
2:  cfg_old ← constructControlFlowGraph(m_old)
3:  cfg_new ← constructControlFlowGraph(m_new)
4:  stack ← new Stack⟨Pair⟨Statement, Statement⟩⟩
5:  stack.push(cfg_old.entry, cfg_new.entry)
6:  while stack is not empty do
7:      ⟨stmt_old, stmt_new⟩ ← stack.pop()
8:      if matchedStmts.keys().contains(stmt_old) || matchedStmts.values().contains(stmt_new) then
9:          continue
10:     end if
11:     matchedPair ← null
12:     if matches(stmt_old, stmt_new) then
13:         matchedStmts[stmt_old] ← stmt_new
14:         matchedPair ← ⟨stmt_old, stmt_new⟩
15:     else
16:         stmtList_old ← BFS(stmt_old, cfg_old, lh)
17:         stmtList_new ← BFS(stmt_new, cfg_new, lh)
18:         ⟨s_old, s_new⟩ ← firstMatchedPair(stmtList_old, stmtList_new)
19:         if ⟨s_old, s_new⟩ ≠ null then
20:             matchedStmts[s_old] ← s_new
21:             matchedPair ← ⟨s_old, s_new⟩
22:         end if
23:     end if
24:     if matchedPair ≠ null then
25:         for each s in BFS(matchedPair.first(), cfg_old, 1) do
26:             for each s' in BFS(matchedPair.second(), cfg_new, 1) do
27:                 stack.push(⟨s, s'⟩)
28:             end for
29:         end for
30:     end if
31: end while
32: return matchedStmts
```

Figure 5.6: Algorithm for matching statements from two methods in ConfSuggester.

*Matching Predicates across Versions*

For each predicate recorded in the old execution trace, ConfSuggester matches it in the new program version to identify its possibly-updated counterpart. The predicate-matching process proceeds in two steps. First, ConfSuggester finds corresponding methods. Then, ConfSuggester matches predicates within matched methods.

To match methods, ConfSuggester uses the first of these two strategies that succeeds:

1. **Identical method name.** Return a method with the identical fully-qualified name in the new version.

2. **Similar method content.** Return the method with the most similar content in the new version. Given a method in the old program version, ConfSuggester uses the algorithm shown in Figure 5.6 (details are discussed below) to match it to *every* method in the new program version, and then chooses the method in the new program version with the most matched statements.

   After running the matching algorithm, ConfSuggester further checks the ratio of matched statements in the old method, and discards method candidates whose matching ratio is below a threshold (default value: 0.9).

If there is no match for the declaring method in the new program version, ConfSuggester concludes that the predicate cannot be matched. Otherwise, ConfSuggester runs the algorithm in Figure 5.6 (or looks up a cached version of the result) to establish the mapping between instructions, and then returns the matched instruction of the predicate (or `null` if the predicate cannot be matched).

**Statement-matching algorithm.** The algorithm in Figure 5.6 is inspired by the JDiff program differencing algorithm [8]. The original JDiff algorithm is based on a method-level representation (called hammocks) that models object-oriented features. It works in a hierarchical way by first identifying matched classes and then matched method pairs, and uses textual similarity to compare two program statements. By contrast, our algorithm directly works on the bytecode, using the program control flow graph representation to establish the matching between statements.

In Figure 5.6, ConfSuggester first constructs the control flow graphs of two given methods (lines 2–3), then pushes their entry nodes (a synthetic node for each method) onto a worklist stack (line 5), which retains the next statement pair for comparison. The algorithm repeatedly pops a statement pair from the stack (line 7) and decides whether the two statements are matched (line 12). Each statement appears at most once in the result (lines 8–9).

The algorithm decides whether two statements are matched by using the matches($s$, $s'$) auxiliary function. Method matches($s$, $s'$) returns true if both $s$ and $s'$ have the same statement type (i.e., the same instruction type in bytecode), and if $s$ and $s'$ are field-accessing or method-invoking statements, the same field or method is accessed or invoked by both statements. Such approximate matching tolerates small differences between two versions, such as changes to constant values.

If two statements are matched, the algorithm saves them in the result map (line 13). Otherwise, the algorithm compares each statement reachable within $lh$ control flow graph edges (lines 16–22). Doing so permits the algorithm to tolerate some small changes in the method code, and attempts to match as many statements as possible.

When two matched statements are found (stored in the $matchedPair$ variable in lines 14 or 21), the algorithm pushes every pair of their successor statements onto the stack (line 27). It terminates after every statement has been attempted to match.

*Identifying Behaviorally-Deviated Predicates*

Using the predicate matching information, ConfSuggester next identifies predicates that behave differently between two versions.

Given an execution trace $T$, ConfSuggester characterizes a predicate $p$'s behavior by how often it is evaluated (i.e., the number of observed executions) and how often it evaluates to true (i.e., the "true ratio"). The true ratio is an important characteristic of a predicate's behavior, but it is less dependable the fewer times the predicate has been executed.

ConfSuggester combines the true ratio and number of executions by computing their harmonic mean.

$$\phi(p, T) = \frac{2}{1/trueRatio(p, T) + 1/totalExecNum(p, T)}$$

In $\phi(p, T)$, $trueRatio(p, T)$ returns the proportion of executions of the predicate $p$ that evaluated to true in $T$, and $totalExecNum(p, T)$ returns the the total number of observed executions of predicate $p$ in $T$. To smooth corner cases, $\phi(p, T)$ returns 0, if a predicate $p$ is not executed in $T$ (i.e., $totalExecNum(p, T) = 0$) or a predicate $p$'s true ratio is 0 (i.e., $trueRatio(p, T) = 0$). We let $\phi(\texttt{null}, T) = 0$ for all $T$.

Given two matched predicates $p_1$ and $p_2$ from two different execution traces $T_1$ and $T_2$, ConfSuggester uses the deviation function defined in Figure 5.8 to compute the behavioral deviation value. In Figure 5.8, the deviation function discards a predicate pair whose behavioral deviation value is less than a pre-defined threshold (line 2). This is for tolerating small non-determinism during program execution and making ConfSuggester focus on predicates with substantial behavioral differences.

The identified behaviorally-deviated predicates indicate different control flow taken between two versions under the same input and configuration. Such control flow differences are useful in explaining which part of the program might be behaving unexpectedly.

### *5.3.4 Configuration Option Recommendation*

In this step, ConfSuggester attributes the control flow differences to one or more root-cause configuration options. The key idea is to identify configuration options that may affect the behaviorally-deviated predicates, and then rank these options by the deviation value (computed by the deviation function in Figure 5.8) and the number of executed statements they control (computed by the `getExecutedStmtNum` auxiliary function in Figure 5.8).

To identify the configuration options that can affect a predicate, a straightforward way is to use program slicing [196] to compute a forward slice from the initialization statement of a configuration option, and then check whether the predicate is in the slice. Unfortunately, traditional full slicing [196] would produce unusably large slices due to its conservatism.

To address this limitation, ConfSuggester uses the same observation used in ConfDiagnoser (Section 4.3.2). and employs thin slicing [182] to identify configuration options that *directly* affect a predicate. Different from traditional full slicing, thin slicing *only* follows the data flow dependencies from the slicing criterion (i.e., the initialization statement of a configuration option) and ignores con-

deviation($p_1$, $T_1$, $p_2$, $T_2$):

1: $result \leftarrow |\phi(p_1, T_1) - \phi(p_2, T_2)|$
   {$\delta$ is a pre-defined threshold with default value: 0.1}
2: **if** $result < \delta$ **then**
3: $result = 0$
4: **end if**
5: **return** $result$

Figure 5.7: The deviation function. It comptues the deviation value between two predicates $p_1$ (from execution trace $T_1$) and $p_2$ (from execution trace $T_2$), and function $\phi$ used in deviation is defined in Section 5.3.3

trol flow dependencies as well as uses of base pointers. Using thin slicing, ConfSuggester separates pointer computations from the flow of configuration option values and naturally connects a configuration option with its affected statements by the data flow dependencies. Section 5.5.3 empirically demonstrates that using traditional full slicing will decrease the accuracy of ConfSuggester.

To reason about the root-cause configuration options, ConfSuggester associates each configuration option with a weight, which represents the strength of the correlative relationship between the configuration option and the execution differences. A larger weight value indicates that a configuration option potentially contributes more to the control flow differences as its value propagates in the program, and thus the configuration option is more likely to be the root cause.

Figure 5.8 presents the configuration option recommendation algorithm. For each behaviorally-deviated predicate in an execution trace, ConfSuggester first attributes the deviated behavior to its affecting configuration options (lines 4 and 12). Then, ConfSuggester computes the number of executed statements controlled by that predicate (lines 5 and 13). To do so, the `getExecutedStmtNum` auxiliary function first statically examines the source code to compute the immediate post-dominator statement [197] of a predicate, and then traverses the execution trace to count the number of statements that are executed between the predicate and its post-dominator statement. ConfSuggester multiples a predicate's deviation value by the number of executed statements, and then updates the weight of each affecting configuration option (lines 5–8 and 13–16). Finally, ConfSuggester ranks all affecting configuration options in decreasing order by weight, outputting a ranked list of suspicious options that might be responsible for the behavioral differences (line 18).

**Auxiliary functions:**
getPredicates($T$): return all executed predicates in the execution trace $T$.
getAffectingOptions($p$, $V$): use thin slicing [182] to compute all configuration options that may affect predicate $p$ in the software version $V$.
getExecutedStmtNum($p$, $V$, $T$): return the number of executed statements (controlled by predicate $p$) in trace $T$ from software version $V$.
**Input**: two software versions: $V_{old}$ and $V_{new}$.
two execution traces: $T_{old}$ and $T_{new}$, on the respective versions.
a map of matched statements between $V_{old}$ and $V_{new}$ : $stmtMap$.
**Output**: a ranked list of likely root-cause configuration options
recommendOptions($V_{old}$, $V_{new}$, $T_{old}$, $T_{new}$, $stmtMap$)

1: $optionMap \leftarrow$ new Map⟨Option, Float⟩
{Each entry of $optionMap$ is initialized to 0.}
2: **for** each $p_{old}$ in getPredicates($V_{old}$) **do**
3: $d \leftarrow$ deviation($p_{old}$, $T_{old}$, $stmtMap[p_{old}]$, $T_{new}$)
4: $options_{old} \leftarrow$ getAffectingOptions($p_{old}$, $V_{old}$)
5: $w \leftarrow d \times$ getExecutedStmtNum($p_{old}$, $V_{old}$, $T_{old}$)
6: **for** each Option $option$ in $options_{old}$ **do**
7: $optionMap[option] \leftarrow optionMap[option] + w$
8: **end for**
9: **end for**
10: **for** each $p_{new}$ in getPredicates($V_{new}$) **do**
11: $d \leftarrow$ deviation($stmtMap^{-1}[p_{new}]$, $T_{old}$, $p_{new}$, $T_{new}$)
12: $options_{new} \leftarrow$ getAffectingOptions($p_{new}$, $V_{new}$)
13: $w \leftarrow d \times$ getExecutedStmtNum($p_{new}$, $V_{new}$, $T_{new}$)
14: **for** each Option $option$ in $options_{new}$ **do**
15: $optionMap[option] \leftarrow optionMap[option] + w$
16: **end for**
17: **end for**
18: **return** $optionMap$.sortedKeys()

Figure 5.8: Algorithm for recommending configuration options. Function deviation is a helper function defined in Figure 5.7.

If two configuration options have the same weights, ConfSuggester prefers the configuration option affecting more statements in its thin slice. This heuristic is based on the intuition that configuration options affecting more statements seem more likely to be relevant to the behavioral differences.

### 5.3.5 Discussion

We next discuss some design issues in ConfSuggester.

***Fixing configuration errors vs. Localizing regression bugs.*** The problem addressed in ConfSuggester is significantly different than the traditional regression bug localization problem [226, 234]. A regression bug occurs when developers have made a mistake, which causes the software to violate its specification after a session of code changes. By contrast, in our context, the software behavior on the new version is still as *designed* by the developers but *undesired* by the users.

***Why not use a dynamic analysis to recommend configuration options?*** ConfSuggester uses thin slicing to statically identify responsible configuration options for a behaviorally-deviated predicate. An alternative is to use a pure dynamic analysis to assess how a configuration option may affect the control flow. Techniques such as Delta Debugging [226], value replacement [238], and dual slicing [186] use a similar idea: they repeatedly replace a variable value with other alternatives, and then re-execute the program to check whether the outcome is desired. There are two major challenges that prevent these dynamic analyses from being used. First, it can be difficult to find a valid replacement value for a non-Boolean configuration option, such as a string or regular expression. Second, automatically checking program outcomes requires a testing oracle, which is often not available in practice, and end-users should not be expected to provide it. To address these challenges, ConfSuggester approximates the program behavioral differences by the control flow differences of two executions, and then statically reasons about the responsible configuration options.

***ConfSuggester's current limitations.*** There are three major limitations in the our ConfSuggester technique. First, ConfSuggester assumes the different behaviors of two program versions are not caused by non-determinism. For non-deterministic behaviors, ConfSuggester could potentially leverage a deterministic replay system [84, 99] to faithfully reproduce the behaviors. Second, ConfSuggester only matches one predicate in the old program version to one predicate in the new program version. If a predicate evolves into multiple predicates in the new version, ConfSuggester may output less useful results. Third, ConfSuggester focuses on identifying root-cause configuration options that can change the functional behaviors of the target program. Configuration options that affect the underlying OS or runtime system, such as the `-Xmx` option used to specify JVM's heap size when launching a Java program, are not supported by ConfSuggester.

## *5.4 Tool Implementation*

ConfSuggester uses the WALA framework [202] to perform offline bytecode instrumentation. The instrumentation code records the execution of every statement and the evaluation result of each predicate. ConfSuggester also uses WALA to analyze Java bytecode statically to identify the affecting configuration options for each predicate that behaves differently across versions.

Like other existing configuration error diagnosis tools [159, 232], ConfSuggester does not instrument libraries such as the JDK, since a configuration option set in the client software usually does not affect the behaviors of its dependent libraries.

## *5.5 Evaluation*

We evaluated 4 aspects of ConfSuggester's effectiveness, answering the following research questions:

1. How accurate is ConfSuggester in identifying the root-cause configuration options? That is, what is the rank of the actual root-cause configuration option in ConfSuggester's output?

2. How long does it take for ConfSuggester to diagnose a configuration error?

3. How does ConfSuggester's effectiveness compare to existing approaches?

4. How does ConfSuggester's effectiveness compare to two variants? The first variant uses full slicing in identifying suspicious configuration options, and the second variant only uses predicate behavior changes to recommend configuration options.

### *5.5.1 Subject Programs*

We evaluated ConfSuggester on 6 Java programs listed in Figure 5.1. The first 5 subject programs are the 5 Java programs studied in Section 5.2, and the remaining subject program is Javalanche [91], which is a mutation testing framework.

We included Javalanche because one of its real users provided us a configuration error he encountered when using Javalanche.

| Program | Old Version | New Version | LOC (new version) | ΔLOC | #Options |
|---|---|---|---|---|---|
| Randoop | 1.2.1 | 1.3.2 | 18571 | 1893 | 57 |
| Weka | 3.6.1 | 3.6.2 | 275035 | 1458 | 14 |
| Synoptic | 0.05 | 0.1 | 19153 | 1658 | 37 |
| JChord | 2.0 | 2.1 | 26617 | 3085 | 79 |
| JMeter | 2.8 | 2.9 | 91979 | 3264 | 55 |
| Javalanche | 0.36 | 0.40 | 25144 | 9261 | 35 |

Figure 5.9: All subject programs used in the evaluation. Column "ΔLOC" shows the number of changed lines of code between the old and new versions. Column "#Options" shows the number of configuration options supported in the new program version.

*Configuration Errors*

For the 5 Java programs studied in Section 5.2, we manually examined all deleted and modified configuration options listed in Figure 5.2. (The added configuration options are unlikely to cause a misconfiguration.) For each change, based on our own understanding, we wrote a test driver to cover it, and then checked whether the test driver could reveal different behaviors on two versions. For those 5 programs, we collected 7 errors as listed in Figure 5.10 (the first 7 errors). For the Javalanche program, we reproduced the reported configuration error. In Figure 5.10, errors #3 and #4 can be reproduced together in a single execution, and each of the other errors is reproduced in one execution.

Our methodology of collecting configuration errors is different from what was used in collecting software regression bugs in the literature [226, 234]. Software regression bugs often can be found in well-maintained bug databases. By contrast, finding recorded configuration errors is much harder, mainly because most configuration errors have not been documented rigorously [222]. Usually, after a session of code changes, when regression tests pass, developers may treat the software behaviors as having been validated. Further, because the software misconfigurations are user-driven, the "fixes" may be recorded simply as pointers to manuals or other documents.

### *5.5.2 Evaluation Procedure*

For each subject program, we used ConfSuggester to instrument both versions. For each configuration error, we used the same input and configuration to reproduce the different behaviors on two instrumented versions.

| Error ID. Program | Error Description | Root-Cause Configuration Option | Num of Options |
|---|---|---|---|
| 1. Randoop | Poor performance in test generation | `usethreads` | 57 |
| 2. Weka | A different error message when Weka crashes | `m_numFolds` | 14 |
| 3. Synoptic | Initial model not saved | `dumpInitialGraphDotFile` | 37 |
| 4. Synoptic | Generated model not saved as JPEG file | `dumpInitialGraphPngFile` | 37 |
| 5. JChord | Bytecode parsed incorrectly | `chord.ssa` | 79 |
| 6. JChord | Method names not printed in the console | `chord.print.methods` | 79 |
| 7. JMeter | Results saved to a file with a different format | `output_format` | 55 |
| 8. Javalanche | No mutants generated | `project.tests` | 35 |
| Average | | | 49.1 |

(a) Configuration errors used in evaluating ConfSuggester.

| Error ID. Program | Rank of the Root-Cause Configuration Option | | |
|---|---|---|---|
| | ConfSuggester | ConfDiagnoser | ConfAnalyzer [159] |
| 1. Randoop | 1 | **N** | **X** |
| 2. Weka | 1 | 9 | 1 |
| 3. Synoptic | 1 | **N** | **X** |
| 4. Synoptic | 6 | **N** | **X** |
| 5. JChord | 1 | 3 | **X** |
| 6. JChord | 1 | **N** | **X** |
| 7. JMeter | 1 | 1 | **X** |
| 8. Javalanche | 3 | 4 | **X** |
| Average | 1.8 | 15.3 | 47.5 |

(b) Experimental results of evaluating ConfSuggester, ConfDiagnoser (Chapter 4), and ConfAnalyzer [159].

Figure 5.10: All configuration errors used in evaluating ConfSuggester and the experimental results. The 2nd error is a crashing error, and all the other errors are non-crashing errors. Column "Root-Cause Configuration Option" shows the actual root-cause configuration option. Column "Num of Options" shows the number of configuration options supported in the new program version, taken from Figure 5.9. Column "Rank of the Root-Cause Configuration Option" shows the absolute rank of the actual root-cause configuration option in each technique's output (lower is better). "**X**" means the technique is not applicable (i.e., requiring a crashing point), and "**N**" means the technique does not identify the actual root cause. When computing the average rank, each "**X**" or "**N**" is treated as half of the number of configuration options, because a user would need to examine on average half of the available options to find the root cause. Column "ConfSuggester" shows the results of using our technique. Columns "ConfDiagnoser" and "ConfAnalyzer" show the results of using two existing techniques as described in Section 5.5.3.

The average size of the execution traces is 40MB, and the largest one (Randoop's trace) is 140MB.

When using ConfSuggester to diagnose a configuration error, we manually specify the initialization statement of each configuration option as the thin slicing criterion. This manual, one-time-cost step took 20 minutes on average per subject program. Future work should automate this manual step. After that, ConfSuggester works in a fully-automatic way: it analyzes two program versions and two

execution traces, and outputs a ranked list of configuration options.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

### *5.5.3 Results*

Figure 5.10 shows the experimental results.

#### *Accuracy*

As shown in Figure 5.10, ConfSuggester is highly effective in identifying the root-cause configuration options that should be changed in the new program version. The average rank of the root cause in ConfSuggester's output is 1.8. For 6 errors, the root-cause configuration option ranks first in ConfSuggester's output; for 1 error, the root-cause configuration option ranks third in ConfSuggester's output; and the root-cause option ranks sixth for the remaining error. ConfSuggester is successful because of its ability to identify the behaviorally-deviated predicates with substantial impacts through execution trace comparison. The top-ranked deviated predicates often provide useful clues about what parts of a program have performed differently.

***Summary.*** ConfSuggester recommends correct configuration options with high accuracy for evolving configurable software systems with non-trivial code changes.

#### *Performance of ConfSuggester*

We measured ConfSuggester's performance in two ways: the performance overhead introduced by instrumentation when demonstrating the configuration error, and the time cost of recommending configuration options. Figure 5.11 shows the results.

The performance overhead to demonstrate the error varies among programs. The current implementation imposes an average 8× and 12.8× slowdown in a ConfSuggester-instrumented old and new program version, respectively. This is due to ConfSuggester's inefficient instrumentation code that monitors the execution of every instruction. The overhead could be reduced by instrumenting at basic block granularity instead. Even so, except for two errors (errors #5 and #6) in JChord, all

| Error ID. Program | Run-time Slowdown (×) | | ConfSuggester time (s) | |
|---|---|---|---|---|
| | Old Version | New Version | Slicing | Suggestion |
| 1. Randoop | 20.1 | 4.1 | 90 | 295 |
| 2. Weka | 1.6 | 1.6 | 80 | 49 |
| 3. Synoptic | 1.7 | 4.7 | 48 | 42 |
| 4. Synoptic | 1.7 | 4.7 | 48 | 42 |
| 5. JChord | 18.7 | 44.3 | 20 | 38 |
| 6. JChord | 17.6 | 41.1 | 23 | 29 |
| 7. JMeter | 1.3 | 1.4 | 51 | 63 |
| 8. Javalanche | 1.4 | 1.5 | 430 | 265 |
| Average | 8.0 | 12.8 | 99 | 91 |

Figure 5.11: ConfSuggester's performance. The "Run-time Slowdown" column shows the cost of reproducing the error in an ConfSuggester-instrumented version of the subject program. The "ConfSuggester time (s)" column shows the time taken by ConfSuggester to diagnose configuration errors in seconds. Column "Slicing" is the cost of computing thin slices on both old and new program versions.

other errors can be reproduced in less than 30 seconds. Errors #5 and #6 require about 20 minutes to reproduce.

ConfSuggester spends an average of 3.1 minutes to recommend configuration options for one error (including the time to compute thin slices and the time to suggest suspicious options). Computing thin slices for all configuration options is non-trivial. However, this step is one-time cost per program and the results can be precomputed. The time used for suggesting configuration options is roughly proportional to the size of the execution trace rather than the size of the subject program.

***Summary.*** ConfSuggester recommends configuration options for diagnosing configuration errors with reasonable time cost.

*Comparison with Two Existing Approaches*

This section compares ConfSuggester with two existing approaches, ConfDiagnoser [232] and ConfAnalyzer [159]. ConfDiagnoser and ConfAnalyzer are among the most precise configuration error diagnosis techniques in the literature.

**ConfDiagnoser**, described in Chapter 4, is an automated software configuration error diagnosis technique. ConfDiagnoser is *not* cognizant of software evolution, and it diagnoses configuration errors from a single program version. ConfDiagnoser assumes the existence of a set of correct

execution traces, which are used to compare against the undesired execution trace to identify the abnormal program parts. When comparing the undesired execution trace with a correct execution trace, ConfDiagnoser only uses a predicate's deviation value to reason about the most suspicious options, while ignoring the statements controlled by a predicate's evaluation result.

To compare ConfSuggester with ConfDiagnoser, we reused the pre-built execution trace databases for the 4 shared subject programs (Randoop, Synoptic, JChord, and Weka) from [232]. Each existing trace database contains 6–16 correct execution traces. For the remaining two subject programs (JMeter and Javalanche), we manually built an execution trace database for each of them by running correct examples from their user manuals. The databases contain 6 and 8 execution traces for JMeter and Javalanche, respectively.

**ConfAnalyzer**, proposed by Rabkin and Katz [159], is a lightweight static configuration error diagnosis technique. ConfAnalyzer tracks the flow of labeled objects through program control flow and data flow, and treats a configuration option as a root cause if its value may flow to a crashing point. Since ConfAnalyzer cannot diagnose non-crashing errors, we can only apply it to diagnose the crashing error in Weka (error #2 in Figure 5.10).

***Results.*** Columns "ConfDiagnoser" and "ConfAnalyzer" in Figure 5.10 show the experimental results. ConfSuggester produces significantly more accurate results than ConfDiagnoser, primarily for two reasons. First, ConfDiagnoser focuses on diagnosing *erroneous* program behaviors and identifies their responsible configuration options. However, for the problem addressed ConfSuggester, the new software version that exhibits *undesired* behavior (after applying the same configuration used in the old version) is working exactly as *designed*. In other words, the execution trace obtained by running the new program version is still *correct*. Therefore, just comparing execution traces obtained from the new program version is not effective in identifying the "abnormal" behavior. By contrast, ConfSuggester compares execution traces from two different versions and directly reasons about the execution differences. Second, ConfDiagnoser only focuses on the predicate behavior changes, while ignoring the statements potentially impacted by the affected predicate. This makes ConfDiagnoser fail to distinguish predicates whose behavioral changes can have different impacts. Section 5.5.3 further evaluates this design choice, showing that considering the number of controlled statements can substantially increase the diagnosis accuracy.

| Error ID. Program | Rank of the Root-Cause Configuration Option | | |
|---|---|---|---|
| | ConfSuggester | Full Slicing | Predicate Behavior |
| 1. Randoop | 1 | 32 | 7 |
| 2. Weka | 1 | 7 | 1 |
| 3. Synoptic | 1 | 16 | 3 |
| 4. Synoptic | 6 | 17 | 8 |
| 5. JChord | 1 | 19 | 5 |
| 6. JChord | 1 | 30 | 5 |
| 7. JMeter | 1 | **N** | 1 |
| 8. Javalanche | 3 | **N** | 13 |
| Average | 1.8 | 20.7 | 5.4 |

Figure 5.12: Experimental results of evaluating two design choices of ConfSuggester. Column "ConfSuggester" shows ConfSuggester's results, taken from Figure 5.10. Column "Full Slicing" shows the results of replacing thin slicing with full slicing in ConfSuggester. "**N**" means the technique does not identify the actual root cause. Column "Predicate Behavior" shows the results of ConfSuggester, if it only considers predicate behavior change. When computing the average rank, each "**N**" is treated as half of the number of configuration options.

ConfAnalyzer outputs the correct result for the crashing error in Weka, but cannot identify root causes for other non-crashing errors. The crashing error in Weka occurs soon after the program is launched. ConfAnalyzer correctly identifies its root cause because a small number of configuration options are initialized and only one of them flows to the crashing point.

ConfSuggester is not directly comparable to other related configuration error diagnosis approaches [12, 14, 184, 203, 209, 219]. Existing approaches target a rather different problem than ConfSuggester, or require different inputs than ConfSuggester. For example, X-Ray [12] diagnoses performance-related configuration errors on a single program version. PeerPressure [203] and RangerFixer [219] only support configuration options defined by certain specific feature models. General software fault localization techniques [102, 132] are not well-suited for configuration error diagnosis, since such techniques often focus on identifying the buggy code or invalid input values. This has been empirically validated in Chapter 4 [232].

***Summary.*** Configuration error diagnosis techniques designed for a *single* program version achieve less accurate results in diagnosing configuration errors introduced in software evolution. ConfSuggester reasons about the behavioral differences between two program versions, and produces more accurate results.

*Evaluating Two Design Choices*

This section evaluates two design choices in ConfSuggester.

**Slicing algorithms.** ConfSuggester uses thin slicing to identify configuration options whose values may affect a predicate. We next evaluate a variant that replaces thin slicing with the traditional full slicing [82]. This variant changes the getAffectingOptions auxiliary function in Figure 5.8, by using full slicing to compute all configuration options that may affect a predicate. Figure 5.12 (Column "Full Slicing") shows the results.

ConfSuggester achieves substantially less accurate results when using full slicing. The primary reason is that full slicing identifies many irrelevant configuration options that *indirectly* affect a predicate of interest. Such configuration options are not pertinent to the task of error diagnosis. Linking them to the exhibited different behavior would degrade ConfSuggester's accuracy. Further, computing full slices is much more expensive than computing thin slices. WALA's full slicing algorithm failed to scale to two subject programs (JMeter and Javalanche).

**Predicate behavioral change metrics.** ConfSuggester considers both the predicate behavior change and the number of affected statements in diagnosing configuration errors. We next evaluate a variant that only uses the predicate behavior change to diagnose errors. This variant changes the `getExecutedStmtNum` auxiliary function in Figure 5.8, by making it always return 1. Figure 5.12 (Column "Predicate Behavior") shows the results.

ConfSuggester's accuracy degrades substantially when ranking predicates based on its behavioral changes without considering the number of affected statements. The primary reason is that behaviorally-deviated predicates occur all over the execution traces, but each predicate may have different impacts to the overall program behavior change. ConfSuggester uses the number of statements determined by the predicate evaluation result to approximate such potential impacts.

***Summary.*** Full slicing includes too many irrelevant program statements due to its conservatism and only using a predicate's behavior change is not enough to identify the root-cause configuration options. ConfSuggester, using thin slicing and considering both the predicate behavior change and the impacted statements, is a better choice in diagnosing configuration errors.

### *5.5.4 Discussion*

***Threats to validity.*** There are several threats to validity of our evaluation. First, the 6 Java programs might not be representative, though some of them have been used in previous research. Likewise, the 8 configuration errors might not be representative, even though we evaluated every error we found. We only evaluated ConfSuggester on errors caused by one configuration option. It is unclear whether ConfSuggester would produce useful results if fixing a particular configuration error requires changing values of two dependent configuration options. Second, our evaluation focused on configuration errors rather than software regression bugs, as all regression tests between two versions pass. We have not evaluated whether ConfSuggester would help users work around buggy program versions. Third, ConfSuggester's effectiveness depends on the effectiveness of the predicate-matching algorithm. In our experiments, on average 12% of lines are changed between the program versions. ConfSuggester may yield less useful results for programs with significant code changes. However, different algorithms can be plugged into ConfSuggester. Fourth, our evaluation only compared ConfSuggester with two other approaches. Comparing with other analyses or tools might yield different observations.

***Experimental conclusions.*** We have three chief findings. (1) ConfSuggester is highly effective in diagnosing configuration errors introduced by software evolution; (2) ConfSuggester produces more accurate results than approaches designed to diagnose errors on a single program version; and (3) ConfSuggester outperforms two variants that use full slicing and only a predicate's behavior change in error diagnosis, respectively.

## *5.6 Summary*

This chapter describes ConfSuggester, a technique to help software users to troubleshoot configuration errors. ConfSuggester focuses on errors caused by software evolution, and recommends configuration options whose values should be changed to produce the desired behavior on the new software version. The key idea of ConfSuggester is comparing two execution traces from two different versions and using static analysis to correlate the difference to the root-cause configuration options. In our experiments, ConfSuggester accurately identified the root causes of 8 configuration errors in 6 real-world software systems.

# Chapter 6

# FIXING BROKEN WORKFLOWS FOR EVOLVING GUI APPLICATIONS

This chapter presents a technique, called FlowFixer, to fix broken workflows for evolving GUI applications. GUI broken workflows are another type of software error that a software user can fix. Section 6.1 introduces the broken workflow problem, Section 6.2 details the FlowFixer technique, Section 6.3 describes our implementation, Section 6.4 presents the evaluation, and Section 6.5 concludes.

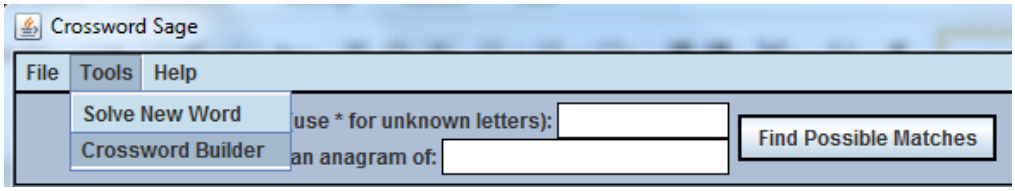
## *6.1 Problem*

Most users interact with a software application through its Graphical User Interface (GUI). The software developers evolve the GUI over time to improve the user experience. Many popular software systems, from web-based systems like social media to desktop applications like office suites, routinely revamp their GUIs. Such GUI changes can be even more frequent than changes to the core domain logic.

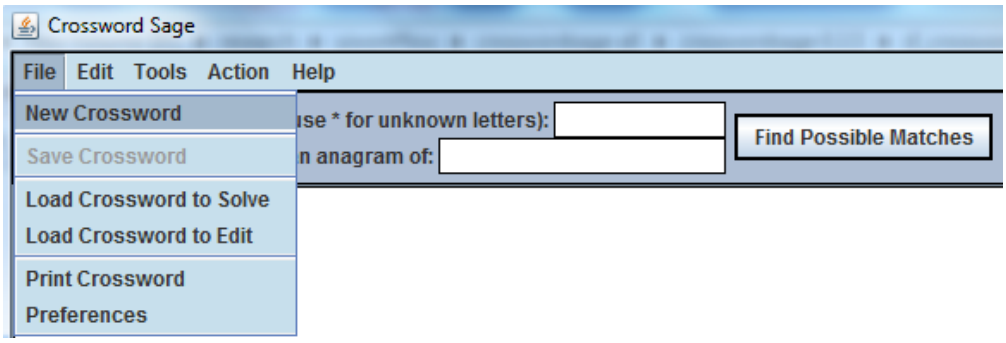
Although application evolution is generally beneficial, it can create usability problems for end-users. Changes ranging from a simple GUI refactoring to a complete rearchitecture can break an end-user's workflow — a sequence of UI actions to complete a specific task. To recover from a broken workflow due to GUI evolution, end-users often need to seek information from online help forums, the software user manual, or experts. This process can be tedious, laborious, and frustrating.

As an example of real GUI application evolution, Figure 6.1 shows screenshots corresponding to versions 0.3 and 0.35 of the Crossword project [37]. The GUI changes break an end-user's workflow. In version 0.3, a user clicks the menu item "Tools $\rightarrow$ Crossword Builder" to create a new crossword puzzle; but in version 0.35, this menu item has been removed, and the user must instead click the menu item "Files $\rightarrow$ New Crossword".

The broken workflow problem is not rare in practice. For example, our examination of Microsoft Office user forums [139] indicates that broken workflows are serious issues. For *every* release of *each*




**(a) Crossword version 0.3.0**




**(b) Crossword version 0.3.5**

Figure 6.1: Evolution of the Crossword [37] GUI breaks an existing workflow: building a new crossword puzzle. In version 0.3, a user clicks "Tools → Crossword Builder" to create a new puzzle. In version 0.35, this menu item has been removed and a functionally equivalent one, "File → New Crossword" is added. Our technique FlowFixer automatically recommends "click menu item: File → New Crossword" as a replacement UI action to repair this broken workflow in version 0.35.

Microsoft Office product in the past 5 years, end-users complained about their broken workflows due to UI changes and sought information to repair them. These users had already tried but failed to find the new workflow before they wrote the forum posting. Writing a good forum post is harder than reproducing an existing workflow (since the post has to clearly describe the existing workflow).

GUI evolution presents problems not just for end-users, but also for software developers. To automate testing of a GUI application, test engineers often write test scripts to mimic end-user workflows by performing actions on GUI elements. Such test scripts are fragile to UI changes. According to an empirical study [136], as many as 74% of test scripts become unusable between successive releases of a typical GUI application. An internal evaluation of automated testing in Accenture showed that even simple modifications to GUIs resulted in 30% to 70% changes to test scripts [67].

Manually repairing every broken workflow is tedious and frustrating, since a GUI application like Microsoft Word contains hundreds of GUI screens and thousands of UI actions. It is infeasible for a software user to explore this space to choose replacement actions.

Another possible approach is to programmatically compare the GUIs of two versions, identify changed GUI elements, and then locate UI actions that reference these modified GUI elements [42, 67, 134]. This is a blackbox impact analysis that does *not* require analysis of the GUI application's source code. Section 6.4.3 empirically demonstrates that existing GUI-comparison-based approaches only repair a small number of broken workflows. There are two fundamental limitations that cause this poor performance. (1) For some UI actions (e.g., filling a blank textbox), their effects cannot be observed by merely comparing GUIs statically without actually executing actions. Existing blackbox GUI-comparison-based approaches fail to distinguish UI actions that look similar but result in different consequences. (2) During software evolution, the GUI can change substantially: a GUI element may be moved from one screen to another, the label of a GUI element may be modified, and a GUI element may even be replaced by another GUI element with a different type (i.e., replacing a menu item with a toolbar button). Therefore, it is generally impractical to find a replacement GUI element and a suitable action on it from the new application version without knowing the precise "action semantics". For these reasons, a GUI-comparison-based approach like [67] only warns about affected workflow steps (i.e., test script statements) that should be modified, but does not indicate how to fix the broken workflow.

## *6.2 Technique*

The FlowFixer approach described in this chapter is a program-analysis-based solution. FlowFixer repairs broken workflows that are affected by GUI evolution by recommending replacement UI actions in the updated GUI application to complete the same workflow.

### *6.2.1 Overview*

The key observation behind FlowFixer is that during the evolution of a GUI application, the underlying system that implements a given functionality often stays relatively the same between versions, even when its GUI evolves rapidly. To repair a broken workflow, FlowFixer analyzes how

the methods invoked by a workflow in the old version were changed during application evolution. Specifically, if a UI action (e.g., click a menu item) is removed from the application, FlowFixer first identifies methods invoked when performing that UI action in the old version, then analyzes how these invoked methods differ in the new version, and finally reasons about which UI actions in the new version can trigger the updated methods.

FlowFixer takes as input an old workflow (that is valid on the old version but broken on the new version), the source code of two versions of a GUI application, and the version history between these two versions. It works in four steps (illustrated in Figure 6.2):

1. **Dynamic Workflow Profiling.** FlowFixer instruments the old application version, so that executing the instrumented application produces an execution trace. FlowFixer asks the user to demonstrate a workflow on its GUI. This is an easy, fast task for the user. FlowFixer analyzes the obtained execution trace to extract all invoked methods.

2. **Static Method Matching.** For each method invoked by the workflow in the old version, FlowFixer identifies its corresponding, possibly updated method in the new version.

3. **Random Action Execution.** FlowFixer instruments the new application version. Then, it randomly executes UI actions on the instrumented version and observes which methods get invoked. Based on the action execution and method invocation information, FlowFixer builds a map from each executed UI action to its invoked methods. This step can be performed ahead of time, by the software vendor.

4. **Replacement Action Recommendation.** For each matched method (from step #2) in the new version, FlowFixer queries the built map (from step #3, mapping each executed UI action to its invoked methods) to find UI actions that can trigger it. FlowFixer ranks these found UI actions, in terms of adapting the broken workflow to the new application.

### *6.2.2 Profiling a Broken Workflow*

FlowFixer first instruments the old version of the tested application offline by inserting code to monitor each method's execution at run time. Then, FlowFixer asks the end-user to demonstrate a

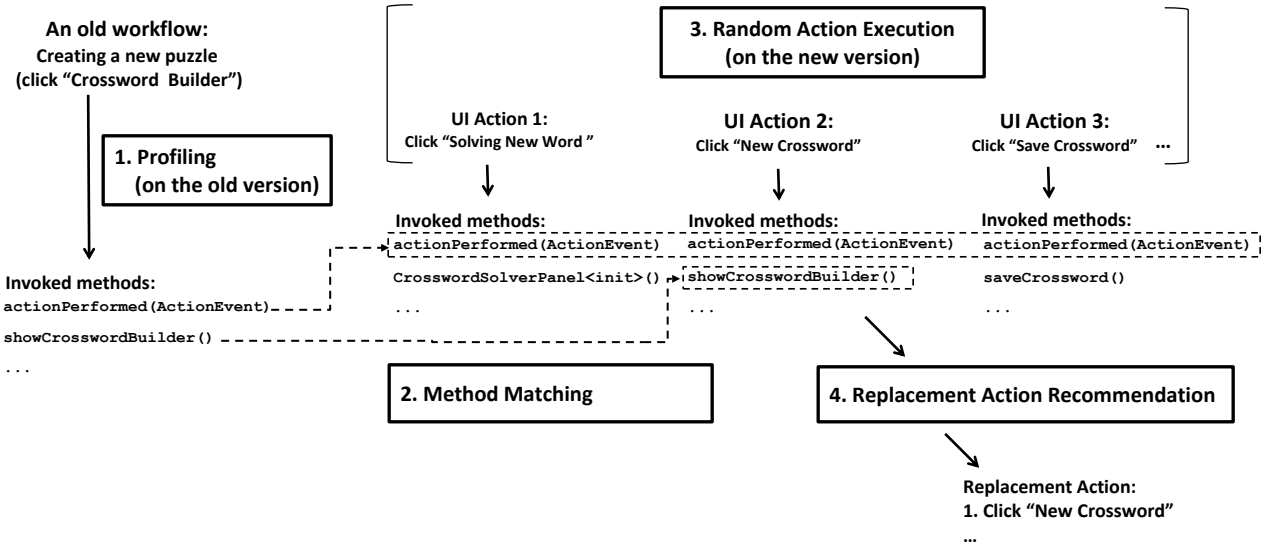



Figure 6.2: Illustration of FlowFixer's 4 steps in repairing a broken workflow using the Crossword example from Figure 6.1. For the Crossword example, FlowFixer's top recommendation to repair the broken workflow is the UI action: click "New Crossword".

workflow on the instrumented version up to the broken action. Demonstration is one of the simplest ways for an end-user to describe a workflow; it is easier than writing specifications or scripts of any form.

During instrumentation, FlowFixer distinguishes event handlers (i.e., event-handling methods) from other methods, and records both of them in the execution trace. An event handler is a special method in a GUI application that is called back by the GUI framework. In a Java application, each feasible UI action is handled by an event handler (possibly shared with other UI actions). The event handler further calls other methods to realize the desired functionality.

Executing the instrumented application produces an *execution trace*, which consists of a sequence of invoked methods. Take the Crossword program in Figure 6.1 as an example: an end-user clicks the menu item "Tools → Crossword Builder" to create a new puzzle. (Here, clicking "Tools → Crossword Builder" is a single GUI action, not two actions.) As shown in Figure 6.2, this old workflow invokes an event handler called `MenuListener.actionPerformed(ActionEvent)` (for short, `actionPerformed(ActionEvent)`), which further calls `showCrosswordBuilder` and other methods (omitted in Figure 6.2). FlowFixer records those invoked methods in an execution

trace file.

### *6.2.3 Matching Methods across Versions*

For each invoked event handler and other methods invoked by the broken workflow in the old version, FlowFixer finds its corresponding, possibly updated method in the new version.

Previous work has described how to match program elements across program versions [39, 47, 111, 137, 157]. We were not able to use an existing tool since the tool implementation is either unavailable [157] or does not support the latest Java version [39], or the technique itself focuses on identifying changes by a small set of refactorings [47, 111] rather than finding the counterpart of an arbitrary program element. Thus, we created our own tool to perform method matching. If other techniques or tools improve, we could integrate them into FlowFixer.

The key to our approach is our observation that during a GUI application's evolution, code implementing the *same* functionality across two versions often stays relatively the same. This is particularly true for event handlers. When developers revamp a GUI, they often *reuse* an existing event handler and make it respond to a different UI event rather than re-writing the same event-handling logic from scratch. Based on this observation, FlowFixer employs three simple heuristics for method matching and uses the first one that succeeds.

1. **Identical Method heuristic.** Return a method with the identical fully-qualified name in the new version.

2. **Similar Name heuristic.** Return all methods with a fully-qualified name whose Levenshtein string similarity [174] to the original method name is within a pre-defined threshold (default: 0.9). This heuristic handles code evolution such as refactoring (e.g., method renaming, and method pull-up).

3. **Co-evolving heuristic.** If a method is replaced by another method in the new version, the deleted method and the replacement methods are often committed in the same revision. This heuristic leverages this observation and parses the application evolution history to identify co-changed methods with the deleted one.

**Input**: a Java GUI application $P$
**Output**: a map from UI action to its invoked methods
exploreGuiApplication($P$)
1: **return** exploreUIScreen(getInitialScreen($P$))

**Input**: a screen $S$
**Output**: a map from UI action to its invoked methods
exploreUIScreen($S$)
1: $actionMap \leftarrow$ new Map$\langle$Action, Methods$\rangle$
2: $actionList \leftarrow$ getAvailableActions($S$)
3: **while** $actionList$ is not empty **do**
4: $action \leftarrow$ removeRandomElement($actionList$)
5: **if** $action$ is not applicable **then**
6: **continue** $\{action$ is disabled or on a different UI element$\}$
7: **end if**
8: $methods \leftarrow$ executeAction($action$)
9: $actionMap$.put($action$, $methods$)
10: **if** popUpModalDialog($S$) **then**
11: $dialog \leftarrow$ getModalDialog($action$)
12: $nextMap \leftarrow$ exploreUIScreen($dialog$)
13: $actionMap \leftarrow actionMap \cup nextMap$
14: **end if**
15: $newActions \leftarrow$ getNewActions($S$)
16: $actionList$.addAll($newActions$)
17: **end while**
18: dispose($S$)
19: **return** $actionMap$

Figure 6.3: Algorithm for randomly executing UI actions on a GUI application.

In Figure 6.2, the Identical Method Heuristic works for the Crossword program. Section 6.4 gives examples where the other heuristics are necessary.

Given a method in the old version, it is possible that this step will return multiple methods in the new version as the *potentially* matched ones. This does not present problems for FlowFixer, since many of the matched methods identified by heuristics are not invoked by any UI actions and thus are automatically ignored. Section 6.2.5 describes the detailed action recommendation algorithm.

### *6.2.4 Executing Random UI Actions*

After identifying the matched methods in the new application version, FlowFixer needs to find

**Auxiliary methods:**
getBrokenHandler($trace$): returns the first event handler recorded in the execution $trace$ that is broken in the new application

getMatchedMethods($V_{old}$, $V_{new}$, $method$): returns a list of methods (including handlers) that match $method$ (Section 6.2.3)

getActions($actionMap$, $methods$): reverse query on $actionMap$ (produced by the algorithm in Figure 6.3); returns actions that invoke any one of the $methods$

getInvokedMethods($handler$, $trace$): returns a list of methods that are invoked by $handler$ in the execution $trace$; that is, the methods invoked after $handler$ but before the next handler in the trace

**Input**: two versions of the GUI application: $V_{old}$ and $V_{new}$;
the execution trace $T$ of a workflow on $V_{old}$; and $actionMap$,
the precomputed result of exploreGuiApplication($V_{new}$) (defined in Figure 6.3)

**Output**: a ranked list of replacement UI actions in $V_{new}$
recommendReplacementActions($V_{old}$, $V_{new}$, $T$, $actionMap$)

1: $handler \leftarrow$ getBrokenHandler($T$)
2: $matchedHandlers \leftarrow$ getMatchedMethods($V_{old}$, $V_{new}$, $handler$)
3: $handlerActions \leftarrow$ getActions($actionMap$, $matchedHandlers$)
4: **if** $|handlerActions|$ = 0 **then**
5: **return** $handlerActions$
6: **end if**
7: $weightMap \leftarrow$ new Map$\langle$Action, Float$\rangle$
8: **for** each $action$ in $handlerActions$ **do**
9: $weightMap[action] = \frac{1}{|handlerActions|}$
10: **end for**
11: $invokedMethods \leftarrow$ getInvokedMethods($handler$, $T$)
12: **for** each $method$ in $invokedMethods$ **do**
13: $matchedMethods \leftarrow$ getMatchedMethods($V_{old}, V_{new}, method$)
14: $actions \leftarrow$ getActions($actionMap$, $matchedMethods$)
15: **for** each $action$ in $actions$ **do**
16: $weightMap[action] = \max(weightMap[action], \frac{1}{|actions|})$
17: **end for**
18: **end for**
19: **return** $weightMap$.sortedKeys()

Figure 6.4: Algorithm for recommending replacement UI actions for a broken workflow.

UI actions that can trigger those matched methods. The new application's GUI can be dramatically different than the old one; thus, we cannot assume the end-users are familiar with it nor ask an end-user to explore every possible UI action on it.

To obtain information about the new application version, FlowFixer employs *random testing* to generate and execute *random UI actions* on the new version, and observes the invoked event handlers and methods in the background.

FlowFixer's random testing monitors the GUI application's state and repeatedly adds newly-available UI actions to the action queue. When random testing a GUI application, the UI action space to be explored can be large. For the sake of efficiency, our algorithm approximates the exploration by executing each UI action (at most) once. This is because, in most cases, a UI action invokes the same event handler even when it exhibits different behaviors in different contexts.

Figure 6.3 sketches the random execution algorithm. The main procedure `exploreGui-Application` starts processing at the initial screen of a GUI application. The `exploreUIScreen` procedure creates a worklist of all available UI actions (line 2). For each action that is applicable (i.e., not disabled) to the current application state, the algorithm executes it and records its invoked methods (lines 8–9). For each applicable action that requires user inputs, the `executeAction` procedure randomly chooses values from a pre-defined value pool. If the executed action creates a modal dialog, the algorithm recursively explores all available actions on the new modal dialog (lines 10–14). After executing each UI action, the algorithm adds newly-available actions (including actions that were ignored on line 5 if they become applicable again) to the worklist (lines 15–16), since executing a UI action may change the program state and enable other actions. For example, in the Crossword program, executing the action of creating a new puzzle enables a new action of saving a puzzle. After executing all applicable actions on a screen, the algorithm disposes that screen (line 18).

Our implementation of `getAvailableActions` ignores several kinds of UI events, such as key pressing, mouse moving, and window disposing. This was not a problem in our experiments (Section 6.4), for three reasons. First, some ignored events such as key pressing often serve as a shortcut for other actions supported by FlowFixer. Second, some ignored events such as mouse moving are usually not essential in a workflow, or if essential are not replaceable by other events. Third, other events such as window disposing are performed by the GUI framework; an end-user rarely uses them in his/her own workflows.

Our implementation of `executeAction` bounds each action execution to 5 seconds; thus, our random testing algorithm always terminates.

Random testing is easy to implement, scalable to large programs, and remarkably effective in practice, including in our context. However, it has no coverage guarantee. Executing random UI actions is not the only way to identify invoked methods for each UI action. Another possible way is to use static analysis to find all reachable methods for a UI action. Compared with random testing, static analysis can be sound but suffers from several limitations. First, it is hard for a static analysis to distinguish UI actions that share the same event handler. For example, in the Crossword program [37], 12 menu items share the same event handler `MenuListener.actionPerformed`. Thus, it is hard for a static analysis to decide which action has been performed when it gets invoked. Second, static analysis is conservative: it may include methods that are actually unreachable in practice and thus introduce noise. Section 6.4.3 measures the coverage achieved by random testing in our evaluation; and Section 6.4.3 empirically compares FlowFixer with an alternative approach based on static analysis, and shows that FlowFixer yields significantly better results.

### *6.2.5 Recommending Replacement UI Actions*

For a broken UI action in a workflow, FlowFixer recommends a list of replacement UI actions that may complete the same workflow in the new version. The basic idea of FlowFixer's replacement UI action recommendation algorithm is to check each matched method in the new version and then infer which UI action is most likely to invoke it. The algorithm described in this section suggests one fix action for one broken action in a workflow rather than for a sequence of actions. If multiple UI actions in a workflow are broken caused by the GUI evolution, the algorithm can be used iteratively. This was not necessary in our experiments.

Figure 6.4 sketches the recommendation algorithm. The algorithm first analyzes a recorded execution trace to extract the event handler invoked by the broken UI action in the broken workflow (line 1). In a Java application, each UI action is handled by an event handler. Since the input execution trace is produced by the user after demonstrating the workflow up to the broken action (Section 6.2.2), the last recorded event handler is the event handler invoked by the broken UI action. Then, the algorithm identifies a list of UI actions in the new version that can trigger the matched event handlers (line 3). If there is no action in the action list, the algorithm concludes that the broken workflow cannot be repaired (lines 4–6). Otherwise, at least one UI action triggers the matched event handler;

and the rest of algorithm deals with this case.

To estimate the likelihood of each UI action being a replacement action, the algorithm associates each UI action with a weight (line 7). The larger the weight, the more likely the action is the desired one. The algorithm identifies program elements relevant to the broken workflow (handlers on line 2, methods on line 11). The weight of a UI action is inversely proportional to the number of UI actions that trigger the program element. If a relevant method is *uniquely* invoked by one action, that action is more likely to be the desired action. On the other hand, if a relevant method is invoked by multiple actions, the likelihood of each action being the desired action decreases. Finally, the algorithm ranks the UI actions by their weights in a decreasing order and returns them (line 19). If two actions have the same weight, the algorithm ranks the action invoking more methods higher in the output.

For example, in Figure 6.2, the matched event handler `actionPerformed` is invoked by three UI actions on the new version, so each action's weight is 1/3. The matched method `showCrossword-Builder` is uniquely invoked by the UI action "Click New Crossword", so its weight is 1. Thus, the algorithm ranks the action "Click New Crossword" highest.

If a replacement UI action is applicable on a window other than the current one, FlowFixer's recommendation also shows the action that triggers the other window.

Our algorithm focuses on the "uniqueness" of a method being invoked by UI actions. An alternative is to rank each replacement action based on coverage: compute the methods covered by the broken UI action, then rank replacements according to what fraction of those methods each replacement covers. Section 6.4.3 empirically compares this approach with FlowFixer's algorithm, and finds that FlowFixer's algorithm yields better results.

### *6.2.6 Discussion*

We next discuss some design issues in FlowFixer.

**Soundness and completeness.** The FlowFixer technique is neither sound nor complete. The primary reason is that both the method matching algorithm and the random testing technique used in FlowFixer are based on heuristics. The method matching algorithm cannot guarantee to identify the counterpart of each invoked method; and the random testing algorithm cannot guarantee to find all invoked methods by a UI action. Despite such limitations, as demonstrated in Section 6.4, FlowFixer is still

useful in repairing many real-world broken workflows.

**Repairing broken GUI test scripts.** Repairing broken GUI test scripts is related to, but more challenging than, repairing broken workflows. A test script can be written in a programming language. This gives it more flexibility than a workflow, for instance, permitting it to perform an action on a disabled GUI element by directly calling its underlying event handler. In addition, a test script may include other computations besides UI actions, and repairing a broken test script requires updating computations as well as UI actions. A GUI test script may be more fragile to software evolution, if it precisely encodes the position of each target GUI element. Therefore, a tiny UI change like switching the positions of two neighboring menu items can break a GUI test script, but won't affect a workflow from an end-user's perspective. Due to these difficulties, existing test script repairing techniques [67, 85, 134] fix "execution errors": they identify statements affected by GUI changes or delete unusable actions to make an obsolete test script executable, while ignoring its original semantics. By contrast, FlowFixer aims to preserve the semantics of a broken workflow. FlowFixer can be viewed as a first step toward solving the more general GUI test script repair problem.

### *6.3 Tool Implementation*

We implemented FlowFixer using the WALA framework [202] and the UISpec4J library [199]. FlowFixer uses WALA to perform offline instrumentation of Java bytecode. To execute UI actions and gather method invocation information, FlowFixer employs UISpec4J to transparently intercept windows in the background without actually rendering the GUI. Currently, FlowFixer supports GUI applications using the Swing framework [95], and supports UI actions on most of the built-in Swing GUI elements, such as button, menu, checkbox, tree structure, listbox, etc. When executing UI actions, FlowFixer employs several simple heuristics to skip actions on GUI elements whose label contains "Exit" or "Quit", since such UI actions often cause the whole application to abort. Although our current implementation only supports GUI applications using the Swing framework, there are no fundamental limitations that prevent the technique itself from being extended to other frameworks.

To increase robustness, when executing a UI action, FlowFixer spawns a UI action execution thread. If the thread hangs before a user-specified time limit (default: 5 seconds), FlowFixer

terminates the thread and executes the next UI action.

## *6.4 Evaluation*

We evaluated four aspects of FlowFixer's effectiveness, answering the following research questions:

1. How accurate is FlowFixer in repairing broken workflows for real-world GUI applications? That is, what is the rank of the actual replacement UI actions in FlowFixer's output?

2. How long does it take for FlowFixer to repair a broken workflow?

3. How does FlowFixer's effectiveness compare to an existing approach based on GUI comparison?

4. How does FlowFixer's effectiveness compare to an alternative approach based on static analysis?

### *6.4.1 Subject Programs and Broken Workflows*

We evaluated FlowFixer on 5 Java programs shown in Figure 6.5. We selected these programs because they are popular open-source, GUI-based applications available at SourceForge.net. Each program contains multiple GUI screens and many GUI elements, and has evolved over a long period of time (5–12 years). In addition, three of the programs (Crossword, Freemind, and Gantt Project) have been used in the GUI testing literature [134, 138].

For each application, we performed a retrospective analysis to select two versions that contain significant GUI changes. We collected all documented workflows from the old version's user manual. Workflows documented in the user manual, though by no means completely covering an application's functionality, are often among the most useful and important ones. Then, we tried to perform each collected workflow on the new version to check whether the workflow broke or not. We considered a workflow broken if a GUI element used in the old workflow did not appear in the new version on the same panel (e.g., the same menu or dialog). Location changes, such as swapping the order of two menu items, do not count as breaking a workflow: even though they may affect a GUI test script [42],

| Subject program | Description | Size of new version LOC | #Class | #Method | Versions | ΔLOC | Broken workflows |
|---|---|---|---|---|---|---|---|
| Crossword [37] | A tool for solving crosswords | 3087 | 19 | 193 | 0.3 → 0.35 | 1386 | 1. Create a new puzzle |
| JEdit [96] | A cross-platform text editor | 32607 | 275 | 1382 | 2.5 → 2.6 | 5017 | 2. Show the previous/next file |
| Gantt Project [62] | A tool for project scheduling and management | 55009 | 771 | 3852 | 2.0.1 → 2.5.4 | 3777 | 3. Zoom in/out the current chart<br>4. Go to the previous/next date<br>5. Save the current chart<br>6. Show critical path<br>7. View chart options |
| JabRef [88] | A tool for bibliography reference management | 83447 | 636 | 3340 | 2.0 → 2.8.1 | 38992 | 8. Search for a record<br>9. Import records into a new database<br>10. Export records from the current database |
| Freemind [57] | A tool for writing mind maps | 70430 | 354 | 3930 | 0.71 → 0.80 | 10757 | 11. Export a map as HTML<br>12. Export all map branches as HTML<br>13. Show the previous/next graph<br>14. Zoom in/out the current graph<br>15. Change a graph node property<br>16. Change a graph edge property |

Figure 6.5: Subject programs and broken workflows used to evaluate FlowFixer. Column "ΔLOC" shows the number of changed lines of code between the old and versions, as counted by UNIX diff. Column "Broken workflows" lists unique broken workflows.

they are unlikely to confuse an end-user. We evaluated every broken workflow we found; we did not select only workflows on which FlowFixer works well. The workflows are of reasonable complexity: each workflow contains 1 to 8 steps with one broken action, and some are non-trivial for a human to fix (see Figure 6.7 for an example).

Overall, there were 356 workflows in the user manuals, and 70 of these (20%) were broken in the new version. Some of the broken workflows had the same root cause: (1) Two workflows might use the same UI element that was changed. (2) If all items of one menu are moved to another, only one broken workflow related to those two menus is listed. (3) A change in the way zooming is done appears only once per application, even though zooming in and out are distinct UI actions.

There were 16 root causes in total. Figure 6.5 lists one representative broken workflow for each root cause.

### *6.4.2 Evaluation Procedure*

For each subject program, we first used FlowFixer to instrument both old and new versions based on the strategy described in Section 6.2.2. We demonstrated each broken workflow on the instrumented old version to obtain an execution trace. After that, FlowFixer analyzed the obtained execution trace and two program versions. FlowFixer works in a fully automatic way to repair a given workflow, and has two possible outcomes: it either recommends a ranked list of replacement actions or concludes that the broken workflow cannot be repaired. For both cases, we manually examined FlowFixer's output to determine its correctness.

We made a simple change to each subject program by removing the splash screen. This is a pure implementation consideration, because the UISpec4J library cannot handle splash screens before a GUI application launches. The code edit affected 10 lines of code in total across the 5 applications, and it did not modify any GUI functionality.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory, running Windows 7.

### *6.4.3 Results*

*Accuracy*

As shown in Figure 6.6, FlowFixer is highly effective in repairing broken workflows. FlowFixer successfully repaired 15 broken workflows. For 13 workflows, the correct action was FlowFixer's first suggestion. For 2 workflows, the correct action was FlowFixer's second suggestion. For 1 workflow, no repair was possible (the functionality had been removed from the application), but FlowFixer incorrectly output a list of possible replacement UI actions.

We use a broken workflow from the Gantt Project program as an example to illustrate FlowFixer's effectiveness. As shown in Figure 6.7, there are significant GUI changes between Gantt Project versions 2.0.1 and 2.5.4. After the GUI evolution, the toolbar button used to save the current chart state in version 2.0.1 (Figure 6.7(a)) disappeared in version 2.5.4 (Figure 6.7(b)). In version 2.5.4, the UI action to save the current chart state is changed from "click a button" to "fill a table cell". The difficulty of finding this replacement action is further exacerbated by the fact that the table to receive the desired UI action is in a different dialog that only becomes visible after users

| Program | Workflow ID | FlowFixer Rank | Root Cause |
|---|---|---|---|
| Crossword | 1 | 1 | A menu item is re-named and moved to another menu |
| JEdit | 2 | 1 | A menu item is moved to another menu |
| Gantt Project | 3 | 1 | A toolbar icon is replaced by button |
| | 4 | 1 | A toolbar icon is replaced by button |
| | 5 | 2 | The original UI action is replaced by a different action (see Figure 6.7 for details) |
| | 6 | 1 | An icon is replaced by a button |
| | 7 | **X** | This functionality is removed |
| JabRef | 8 | 1 | A menu item is moved to another menu |
| | 9 | 1 | A menu item is re-named |
| | 10 | 2 | A menu item is re-named |
| Freemind | 11 | 1 | A menu item is re-named and moved to another menu |
| | 12 | 1 | A menu item is re-named and moved to another menu |
| | 13 | 1 | A menu item is moved to another menu |
| | 14 | 1 | A menu item is moved to another menu |
| | 15 | 1 | A menu item is moved to another menu |
| | 16 | 1 | A menu item is moved to another menu |

Figure 6.6: Experimental results in repairing broken workflows. Column "Rank" shows the absolute rank of the actual replacement UI action in FlowFixer's output (lower is better), in which "**X**" means FlowFixer output a list of replacement UI actions but the workflow is actually un-repairable. Column "Root Cause" shows the root cause of the broken workflow.

click another button. For this example, FlowFixer outputs the desired UI action as the second suggestion in its report. The key fact explored by FlowFixer during repair is that a method named `UndoableEditImpl.createTemporaryFile`, invoked by the broken workflow on the old version, exists in the new version and is uniquely invoked by the "table cell filling" UI action.

One broken workflow in Gantt Project is not repairable in the new version. In version 2.0.1, Gantt Project provided a menu item called "Chart options" as a shortcut to configure the current Gantt chart, but this menu item was removed in version 2.5.4. We checked the source code of version 2.5.4, and confirmed that code implementing this shortcut was no longer reachable by any event handler. For this workflow, FlowFixer produces a list of replacement UI actions because the Method Matching step (Section 6.2.3) returns a matched method in the new version that is not actually relevant. Future work should remedy this problem. One possible way is to improve the precision of the method

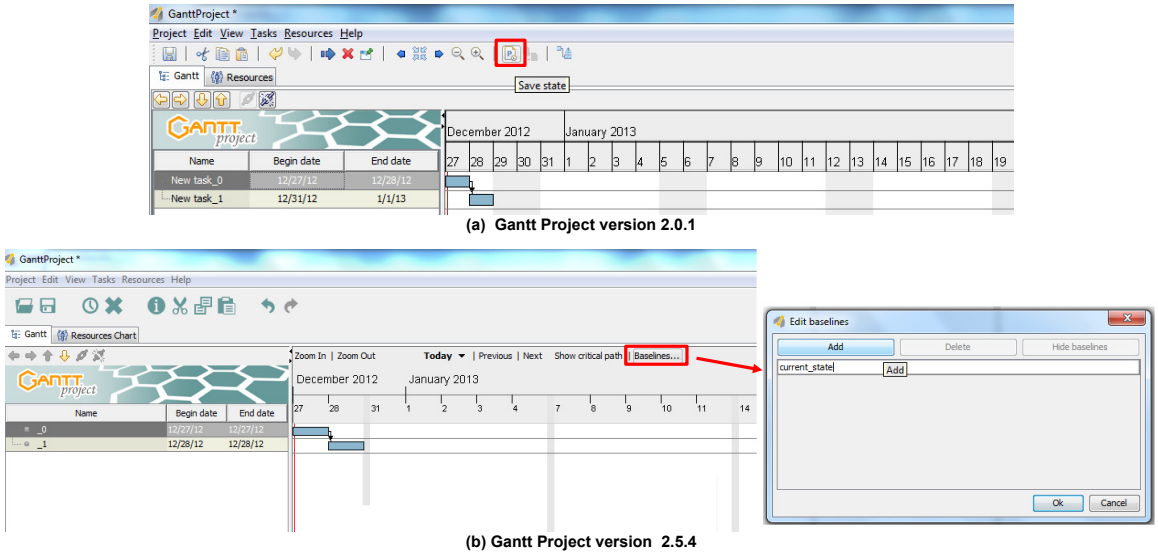



Figure 6.7: The GUI change between two Gantt Project versions breaks workflow #5: saving the current chart state. In version 2.0.1, users click a toolbar button called "Save state" (highlighted in (a)) to complete this workflow. However, in version 2.5.4, the GUI change forces users to first click a toolbar button "Baselines..." (highlighted in (b)), then input a name in the table cell in the popup modal dialog to save the current state. Our FlowFixer technique can suggest this replacement action to repair the broken workflow.

| Number of workflows that FlowFixer can repair | | |
|---|---|---|
| Identical Method | Identical Method + Similar Name | Identical Method + Similar Name + Co-evolving |
| 11 | 13 | 15 |

Figure 6.8: Number of broken workflows that FlowFixer can repair using the three heuristics in Section 6.2.3.

matching algorithm by further analyzing a method's calling context [39].

Figure 6.8 shows the effects of using different method matching heuristics (Section 6.2.3) in repairing broken workflows. Our experimental results confirm the observation that the underlying code implementing the same functionality often stays relatively the same between versions, even when the codebase evolves substantially. For more than 75% of the workflows in our experiment, two method-name-based matching heuristics are sufficient. Figure 6.2 shows an example of method matching using the Identical Method heuristic. The Similar Name heuristic handles a change in the

| Program | Random UI Action Execution | |
|---|---|---|
| | Event Handler Coverage | Overall Method Coverage |
| Crossword | 80% | 27% |
| JEdit | 52% | 42% |
| Gantt Project | 24% | 26% |
| JabRef | 10% | 20% |
| Freemind | 42% | 26% |
| Average | 42% | 28% |

Figure 6.9: Experimental results in executing random UI actions. Columns "Event Handler Coverage" and "Overall Method Coverage" show the coverage of event handlers and methods (including event handlers) in random UI action execution, respectively.

Gantt Project program, when the method to handle panel scrolling event is changed from `ScrollingManagerImpl.scrollRight` to `ScrollingManagerImpl.scrollBy`. The Co-evolving heuristic permits FlowFixer to identify a matching in the Freemind [57] program, when the event handler for editing a graph edge is changed from `EdgeStyleAction.actionPerformed(ActionEvent)` to `EdgeStyleAction.apply(MapAdapter, MindMapNode)`.

Figure 6.9 shows the experimental results of executing random UI actions. On average, the random UI action execution algorithm described in Section 6.2.4 achieved 42% and 28% coverage for event handlers and overall methods, respectively. The coverage is not high, but still useful for recommending replacement UI actions. The low coverage is because most of the uncovered event handlers and methods require a UI action to satisfy some pre-condition, such as executing on a specific application state, and the random testing algorithm was not able to satisfy the pre-condition. Future work should remedy this problem. One possible approach is to integrate FlowFixer with existing symbolic analyses [75] to construct the required desired application states. Despite this limitation, methods invoked by a UI action even without satisfying all pre-conditions often give enough information to disambiguate different UI actions. Take the Crossword program from Figure 6.2 as an example. The `showCrossworldBuilder` method contains code to perform pre-condition checking. Even though a randomly-generated UI action may fail to bypass the pre-condition checking code and miss other methods called by `showCrossworldBuilder`, knowing an action can invoke method `showCrossworldBuilder` is sufficient to disambiguate it from others. This is because the `showCrossworldBuilder` method is uniquely invoked by the action of clicking "New

Crossword".

We next evaluate an alternative approach to recommend replacement actions. This approach replaces FlowFixer's algorithm in Section 6.2.5 with a ranking heuristic based on the proportion of invoked methods in response to the broken UI action that are covered by each alternative replacement UI action, with higher ranks assigned to actions that cover more methods. This approach degraded the accuracy of FlowFixer for every broken workflow shown in Figure 6.6. The primary reason is that each UI action can invoke many utility methods, so two UI actions sharing a large proportion of (utility) methods do not necessarily indicate that they perform similar tasks. By contrast, FlowFixer's heuristic focuses on the "uniqueness" of a method being invoked by UI actions, and yields better results.

***Summary.*** FlowFixer repairs realistic broken workflows with high accuracy for evolving GUI applications with non-trivial GUI changes.

*Time Cost*

We measured FlowFixer's performance in two ways: the time cost in executing random UI actions for each subject program and the time cost in recommending replacement UI actions for each broken workflow. Figure 6.10 shows the results.

On average, FlowFixer uses 27 minutes to execute random UI actions on each subject program. However, random testing is a one-time cost per program and the computed results can be cached to share across workflows.

FlowFixer spends an average of 3.2 minutes to recommend replacement UI actions for one workflow. The time used to repair a workflow is roughly proportional to the number of methods invoked by a workflow, rather than the size of the subject program.

***Summary.*** FlowFixer repairs realistic broken workflows with acceptable time cost.

*Comparison with a GUI-Comparison Approach*

We compared FlowFixer with an existing approach called REST [67]. We chose REST because it is a recent technique targeting a similar problem. REST is a technique to maintain GUI test scripts. It first compares the GUIs of two application versions to identify changed GUI elements. Then, it

| Program | Workflow ID | Time (seconds) | |
|---|---|---|---|
| | | Action Execution | Recommendation |
| Crossword | 1 | 22 | 37 |
| JEdit | 2 | 3611 | 47 |
| Gantt Project | 3 | 2664 | 205 |
| | 4 | | 324 |
| | 5 | | 353 |
| | 6 | | 299 |
| | 7 | | 319 |
| JabRef | 8 | 335 | 189 |
| | 9 | | 118 |
| | 10 | | 402 |
| Freemind | 11 | 700 | 65 |
| | 12 | | 112 |
| | 13 | | 150 |
| | 14 | | 192 |
| | 15 | | 110 |
| | 16 | | 122 |
| Average | | 1642 | 190 |

Figure 6.10: FlowFixer's performance in repairing broken workflows. The time cost has been divided into two parts: executing random UI actions for each subject program (column "Action Execution") and recommending replacement UI actions for each workflow (column "Recommendation").

localizes GUI test script statements that are affected by the changed GUI elements. REST has a different focus than FlowFixer: it aims to identify affected test script statements rather than repairing an affected test script.

We extended REST to support repairing a broken workflow as follows. For each affected UI action in a broken workflow, we use REST to identify the same GUI element in the new version, and then recommend UI actions on this GUI element as the replacement actions. For instance, if a menu item used in the broken workflow disappears in the new version, we use REST to find a menu item with the same label in the new version as the replacement UI action.

Figure 6.11 shows the experimental results. FlowFixer repaired 16 broken workflows, while the REST-based technique only repaired 6 broken workflows. The primary reason for REST's poor performance is that a GUI can change substantially between two application versions and such non-trivial change is often beyond REST's ability to identify the replacement UI actions in the new version. For example, the text on a GUI element can be modified (e.g., the Crossword example in Figure 6.1), a GUI element can be replaced by a different GUI element (e.g., workflows 3, 4, and 6),

| Program | Workflow ID | Repairable? | Can repair the workflow? | |
|---|---|---|---|---|
| | | | FlowFixer | REST |
| Crossword | 1 | Yes | Yes | No |
| JEdit | 2 | Yes | Yes | Yes |
| Gantt Project | 3 | Yes | Yes | No |
| | 4 | Yes | Yes | No |
| | 5 | Yes | Yes | No |
| | 6 | Yes | Yes | No |
| | 7 | No | No | No |
| JabRef | 8 | Yes | Yes | Yes |
| | 9 | Yes | Yes | No |
| | 10 | Yes | Yes | No |
| Freemind | 11 | Yes | Yes | No |
| | 12 | Yes | Yes | No |
| | 13 | Yes | Yes | Yes |
| | 14 | Yes | Yes | Yes |
| | 15 | Yes | Yes | Yes |
| | 16 | Yes | Yes | Yes |

Figure 6.11: Experimental results in comparing FlowFixer with a REST-based technique [67]. Each cell in columns "FlowFixer" and "REST" shows whether the corresponding technique can repair a broken workflow or not. FlowFixer repairs 15 workflows in total. By contrast, the REST-based technique repairs 6 workflows in total and incorrectly states that 9 workflows are un-repairable. For workflow 7 that is actually un-repairable, FlowFixer incorrectly outputs a list of replacement UI actions, while the REST-based technique correctly states that workflow cannot be repaired. The time cost of the REST-based technique is slightly lower than FlowFixer and is omitted for brevity.

and the action on a GUI element to complete the same workflow can be changed to a different action (e.g., the Gantt Project example in Figure 6.7). The 6 workflows that REST can repair all share the same root cause: a menu item is moved from one menu to another without changing its label.

We did not compare FlowFixer with other related GUI testing [135, 224, 225] and GUI test script repairing approaches [85, 134] because these approaches target a rather different problem than FlowFixer. Automated GUI testing frameworks like GUITAR [135, 224, 225] aim to systematically validate a GUI application's functionality. Test script repairing approaches like [85, 134] aim to repair *unusable* test scripts by simply removing or updating affected statements without preserving the original testing semantics, and thus are inapplicable to broken workflow repair. By contrast, FlowFixer recommends replacement UI actions to complete the *same* workflow. In addition, some techniques [85, 134] only work for GUI tests generated by the techniques themselves. It is unknown whether such techniques can be generalized to support repairing realistic broken workflows.

| Program | Workflow ID | FlowFixer | | Static Analysis | |
|---|---|---|---|---|---|
| | | Rank | Time (s) | Rank | Time (s) |
| Crossword | 1 | 1 | 59 | 7 | 194 |
| JEdit | 2 | 1 | 3658 | 70 | 939 |
| Gantt Project | 3 | 1 | 2869 | 13 | 1951 |
| | 4 | 1 | 2988 | 11 | 2402 |
| | 5 | 2 | 3017 | 25 | 3078 |
| | 6 | 1 | 2963 | 25 | 2414 |
| | 7 | **X** | 2983 | **X** | 2546 |
| JabRef | 8 | 1 | 544 | **N** | 4040 |
| | 9 | 1 | 473 | 22 | 4013 |
| | 10 | 2 | 737 | **N** | 4134 |
| Freemind | 11 | 1 | 765 | 20 | 978 |
| | 12 | 1 | 812 | 19 | 1132 |
| | 13 | 1 | 850 | 21 | 1226 |
| | 14 | 1 | 892 | 22 | 1255 |
| | 15 | 1 | 810 | 14 | 1116 |
| | 16 | 1 | 822 | 13 | 1198 |
| Average | | 1.1 | 1682 | 22 | 2039 |

Figure 6.12: Comparison of FlowFixer with a static analysis-based approach described in Section 6.4.3. Column "Rank" shows the absolute rank of the actual replacement UI action in the output. "**N**" represents that the actual replacement UI action is not in the output. "**X**" represents that the workflow is un-repairable but the tool suggests a replacement UI action. Both "**N**" and "**X**" are dropped when averaging ranks. The "Rank" and "Time (s)" columns for FlowFixer's results are taken from Figure 6.6 and Figure 6.10, respectively. For fair comparison, the time cost of FlowFixer includes both the random UI execution time and the replacement action recommendation time.

***Summary.*** Comparing GUIs of two application versions without analyzing the program is insufficient in repairing realistic broken workflows. A program-analysis-based approach like FlowFixer can produce significantly better results.

*Comparison with Static Analysis*

As described in Section 6.2.4, FlowFixer uses random testing, a typical dynamic analysis, to identify methods that can be invoked by a UI action. Another possible way to do so is to statically examine the source code to identify reachable methods for each UI action. Compared with random testing, using static analysis is conservative and sound: it outputs results that describe the program's behavior, no matter on what inputs or in what environment the program is run. However, a conservative static analysis may output methods that are actually unreachable by any UI action.

To investigate the trade-offs between using dynamic and static analyses, we compared FlowFixer with an alternative approach based on static analysis. The alternative approach replaces the random UI action execution step (Section 6.2.4) with the following static analysis. The static analysis examines the source code to build a call graph for the program and identifies possible event handlers for each GUI element declaration. After that, the static analysis follows the call graph to identify all reachable methods from each event handler and treats them as reachable methods by a UI action.

Figure 6.12 shows the experimental results. The static analysis-based approach produces less accurate results. There are two primary reasons for this. First, an event handler may be shared by many GUI elements. For example, in Crossword, 12 UI actions share the same event handler `MenuListener.actionPerformed`. Thus, without executing the program, it is hard for a static analysis to precisely reason about the correct UI action from an invoked event handler. Second, static analysis is conservative and can include many methods that are actually unreachable by UI actions. This causes the recommendation algorithm (Figure 6.3) to recommend UI actions that cannot actually invoke a matched method at runtime.

As shown in Figure 6.12, the cost of static analysis is comparable to FlowFixer's. The majority of the time was spent traversing the call graph to find reachable nodes for every UI action. The static call graph contains 26639–44407 nodes when using the RTA call graph construction algorithm [15]. We did not use more expensive algorithms like $k$-CFA ($k > 1$), because they do not scale to our subject programs.

***Summary.*** Random testing, though neither sound nor complete, can provide more accurate and useful results than static analysis.

### *6.4.4 Discussion*

***Threats to validity.*** There are several threats to the validity of our evaluation.

The 5 Java programs and their changes might not be representative, though some were used in previous research. Likewise, the 16 broken workflows might not be representative, even though we evaluated every broken workflow we found. For example, each broken workflow we found contains only one broken action. Furthermore, FlowFixer ignores some UI actions, such as mouse moving and key pressing. Thus, we do not claim the results will apply for every workflow and GUI change.

FlowFixer's effectiveness depends on the effectiveness of the method matching algorithm and random testing. It may yield less useful results for GUI programs with significant code changes, or for programs with constrained interfaces (that random testing fails to provide useful results). However, different algorithms can be plugged into FlowFixer.

Our evaluation only compared FlowFixer with two other approaches. Comparing with other analyses or tools might yield different observations.

Our evaluation focuses on FlowFixer's algorithm for workflow repairing. A future user study should evaluate whether FlowFixer helps users.

***Experimental conclusions.*** We have three chief findings. (1) FlowFixer is useful in recommending replacement UI actions to repair broken workflows. (2) Comparing GUIs of versions to identify replacement actions is insufficient in practice. (3) Compared with static analysis, random testing yields more accurate and useful results for the broken workflow repair problem.

### *6.5 Summary*

This chapter presented a practical technique (and its tool implementation, called FlowFixer) for repairing broken workflows for evolving GUI applications. The key idea of FlowFixer is comparing execution traces on two different software versions and correlating the execution trace difference to its root cause — a GUI action that may fix the broken workflow. Our experimental results show that FlowFixer is accurate and efficient.

# Chapter 7

# RELATED WORK

This chapter presents related work on program specification inference (Section 7.1), automated software testing (Section 7.2), and software error diagnosis and patching (Section 7.3).

## *7.1 Program Specification Inference*

Two techniques described in this dissertation are related to research in the field of specification inference: Palus infers *software behavioral models* to guide random test generation (Chapter 2), and FailureDoc generalizes *failure-correcting object properties* to explain a failed test (Chapter 3). Both *software behavioral models* and *failure-correcting object properties* can be seen as specialized instances of software specifications. We next discuss related work on using program analysis to infer software specifications in the literature.

### *7.1.1 Dynamic Inference*

There has been a rich body of work on using dynamic analysis to infer software specification. The concept of learning models from actual program runs was pioneered in [36] by applying a probabilistic NFA learner on C traces. Their approach relies on manual annotations to relate functions to objects and to distinguish object definers from object users.

Ernst et al. [49] have developed Daikon (later refined for object-oriented systems [38]) and Hangal and Lam [76] have developed DIDUCE. These tools discover program invariants by monitoring the runtime state of a program and attempting to match invariant templates to expressions. These tools are able to infer simple properties, like `a`$\neq$`0`, but at present do not infer temporal properties.

Combining the ideas of invariant detection and temporal property mining, Lorenzoli et al. have developed a dynamic analysis algorithm for extracting software behavioral models [129]. The algorithm, GK-tail, builds an extended finite state machine from a set of dynamic traces. The transitions in these extended models include both a called function or method and a set of constraints

on the parameters or environment.

Gabel and Su [59, 61] describe Javart, a specification mining framework that can learn, fully automatically, complex temporal properties from execution traces. The key insight behind Javert is that real, complex specifications can be formed by composing instances of small generic patterns. In particular, Javert learns simple generic patterns and composes them using sound rules to construct large, complex specifications.

Compared to existing specification inference techniques, the call sequence model used in Palus is designed for test generation. It aims to provide general guidance in creating a legal sequence, but does not try to generalize the observations (e.g., inferring temporal properties such as method `f` is always called after method `g`) as many specification inference techniques do. The abstract object profile we used to enhance the original call sequence model (Section 2.3.1 and [236] for details) is closely related to the ADABU model [41] and one of its variants [40]. The difference is that the ADABU model [41] classifies methods into *mutators* (methods that change the object state) and *inspectors* (methods that keep the state unchanged), and then uses the return value of each *inspector* to represent an object state. The abstract object profile used in Palus is more similar to a variant of the original ADABU model [41], which directly maps the value of each mutable fields to an abstract domain for object representation. However, the work described in [40] focuses on using the ADABU model for specification mining and typestate verification, and it is unclear whether the ADABU model can be used for unit test generation. By contrast, Palus uses the abstract object profile to create legal and behaviorally-diverse sequences for testing purposes.

### *7.1.2 Static Inference*

There is a rich body of static software specification inference techniques. Although the literature on software specification inference from source code is varied, many techniques and tools share similar characterizations of properties: finite automata that describe correct behavior.

Ammons et al. [3] develop a specification miner, Strauss, that mines specification by learning a probabilistic finite state automaton. As a typical static analysis, Strauss requires the input alphabet of the automaton to be manually specified.

Whaley et al. [208] present an alternative method to mine models for classes: the user specifies

the input alphabet of the automaton (a Java API), and the algorithm identifies and prunes illegal transitions based on the state-checking code that throws exceptions on errors. Their approach relies on defensively-programmed components that imply correct usage through error checking.

Shoham et al. [173] present a technique based on abstract interpretation that is capable of learning arbitrarily complex specifications. The abstract value is the automaton itself, and it is constructed as the program is interpreted. This technique is accurate, but like the previously mentioned tools, the alphabet of the automaton must be known before the analysis begins.

PR-Miner [120] mines association properties from programs by correlating related function calls using frequent itemset mining. Unlike many static analyses as well as Palus, PR-Miner requires no input from the user other than the program (Palus requires a sample execution trace). However, PR-Miner derives sets of functions, variables and data types that frequently appear together, and does *not* take ordering into account. In other words, the output of PR-Miner is only related by frequent association and not by the order or frequency of invocation and thus is unable to guide a test generator to create legal tests for programs with constrained interfaces.

More recently, Gabel and Su [60] leverage the use of symbolic techniques to expand the tractability of the static specification inference problem and allow the mining of larger method-call sequence patterns. Their work explores the observation that there is a great deal of regularity in the representation and tracking of all possible combinations of system components. Thus, a symbolic algorithm based on binary decision diagrams (BDDs) can exploit this regularity. However, while they show some speed-up, the technique still faces the challenge of scalability, and the length of mined method-call sequence is limited in practice to 3.

Compared to dynamic approaches, static analyses operate on the program text, not on particular software executions, and are typically sound but conservative. However, static analyses suffer from several limitations. First, they may omit properties that are true but incomputable and properties that are too expensive to compute. Second, static analyses are limited by the high cost of modeling certain program structures and states (e.g., loops, recursions, pointers, and heaps). Approximations can be used but often cause a static analysis to produce inaccurate results. For instance, precise loop modelling is still beyond the state of the art; pointer manipulation forces many static analyses to give up or to approximate, resulting in overly weak properties. Third, static analyses often require developers to write annotations or (partial) specifications. This may decrease their usability in

practice. By contrast, Palus and FailureDoc overcome the above limitations by using dynamic analyses (i.e., analyzing a sample execution trace). Compared to static analyses, dynamic techniques can detect context-dependent properties (e.g., method-call sequences in Palus) and can easily check properties that stymie static analyses. Thus, Palus is able to learn useful behavioral models even from a small number of execution traces and FailureDoc can generalize valuable failure-correcting object properties from a single failing execution trace. On the other hand, static analyses are complementary to the dynamic approaches used in Palus and FailureDoc: static analysis examines program code and reasons over all possible behaviors that might arise at run time, but a dynamic tool has a limited view of the targeted program. How to combine their mutual strengths to perform specification inference is an interesting research direction [50].

### *7.1.3 Statistical Inference*

There has been a recent trend in using statistical and machine learning techniques to automatically discover specifications.

The invariant detection algorithm in Daikon [49] is essentially a machine learning algorithm: it uses a generate-and-check algorithm to test a set of pre-defined potential properties against the values, and then reports those properties that are not falsified.

Kremenek et al. [114] use annotation factor graphs to probabilistically assign annotations to functions and form specifications. This technique allows the user to incorporate other domain-specific information into the analysis, like a belief in the overall ratio of allocators to deallocators, as components in the factor graph. While [114] only captures pair-wise programming rules, Li and Zhou [120] improve this technique by mining more complex rules. Using a similar approach, Dynamine [126] identifies highly correlated method calls as well as common bug fixes by mining source code check-ins.

Statistical methods are also be used to reduce the incidence of false positives. For example, Le Goues and Weimer [66] describe a specification miner that leverages a statistical model to drastically reduce the incidence of false specifications that do not hold in practice. Related, Merlin [125] and Anek [18] are two recent probabilistic specification inference tools. Merlin infers explicit information flow specifications from program code. It models information flow paths in a data propagation graph

using probabilistic constraints, and then approximates these path constraints using constraints on chosen triples of nodes, resulting in a cubic number of constraints that are dramatically fewer than the brute-force approach. Anek infers specifications useful for modular typestate checking of programs. Differing from Merlin, the specifications inferred by Anek consist of pre and postconditions along with aliasing annotations known as access permissions.

The property inference component in FailureDoc uses a statistical, Daikon-like algorithm to generalize failure-correcting objects. It is essentially a generate-and-check algorithm to test a set of pre-defined potential properties against the values, and only reports those properties that are valid. Compared to the invariant detection engine implemented in Daikon [49], the property set checked by FailureDoc is highly tailored for the debugging purpose. It discards many scalar properties that are extensively used in Daikon, and adds some properties that Daikon lacks, such as checking whether all objects have the same abstract object profile.

## *7.2 Automated Software Testing*

Four techniques described in this dissertation are related to research in the field of automated testing: Palus generates unit tests (Chapter 2), ConfDiagnoser and ConfSuggester diagnose errors in a configurable system (Chapters 4 and 5), and FlowFixer uses random testing to generate UI action sequences as part of its broken workflow repair algorithm (Chapter 6). We next discuss related work on automated test generation, testing configurable software, and testing GUI applications.

### *7.2.1 Test Generation*

To reduce the burden of manually writing unit tests, many automated test generation techniques have been developed [10, 28, 31, 108, 148, 191]. Of existing test generation techniques, bounded-exhaustive [22, 131], symbolic execution-based [65, 172, 212, 230], and random [28, 108, 148] approaches represent the state of the art.

Bounded-exhaustive approaches [22, 131] generate sequences exhaustively up to a small bound on sequence length. However, generating target test reaching many program states often requires longer sequences beyond the small bound [191].

Symbolic execution-based analysis tools such as JPF [155] and CUTE/jCUTE [172] explore

paths in the program under test symbolically and collect symbolic constraints at all branching points of an explored path. The collected constraints are solved if feasible, and a solution is used to generate an input for a specific method. However, these symbolic execution-based tools face the challenge of scalability, and the quality of generated inputs heavily depends on the test driver (which is often manually written [33, 65, 155, 172, 201]).

Random test generation approaches have been demonstrated to be easy-to-use, scalable, and promising in finding previously-unknown bugs [147, 148]. Tools like JCrasher [28], Jartege [145], AutoTest [31], Eclat [146], and Randoop [148] generate method-call sequences for object-oriented programs using various strategies and information obtained from programs. For example, JCrasher [28] creates test inputs by using a parameter graph to find method calls whose return values can serve as input parameters. Eclat [146], utilizes top-down and bottom-up strategies to generate random sequences of method-calls, and classifies them as normal, illegal, or error-revealing. Randoop [148] uses test execution result as feedback information to guide the test generation procedure. However, pure random test generation techniques face challenges in achieving high structural coverage for programs that have constrained interfaces. One major reason is that, for such programs, correct operations require calls to occur in a certain order with specific arguments. Thus, there is often a low probability of generating required sequences at random for achieving target states. On the other hand, evolutionary approaches accept an initial set of sequences and evolve those sequences to produce new sequences that can generate target states. These approaches use fitness [124], a metric computed toward reaching a target state, as a guidance for producing new sequences. However, the generation is still a random process and shares the same characteristics as random approaches discussed above.

To handle the large search space of possible method-call sequences, several data mining-based approaches have been proposed to improve the quality of generated tests. Approaches like [191] mine client code bases statically and extract frequent patterns as implicit programming rules. These approaches use frequent itemset mining or association rule mining for extracting frequent patterns, which are used to assist in generating tests. However, these approaches might be sensitive to a specific client program, and thus the results could heavily depend on the quality of the client code base provided by the code search engine.

Some other approaches improve the effectiveness of automated testing techniques by using a formal specification to guide test generation. For example, tools like Jartege [145] and AutoTest [31]

use a formal specification provided by developers to determine whether the generated method calls are error-revealing. However, such a specification is often absent in practice.

Compared to existing approaches, our Palus technique described in Chapter 2 takes a different perspective to address the method-call sequence generation problem. It first uses dynamic analysis to infer an enhanced call sequence model from a sample execution, then uses static analysis to identify method dependence relations based on the fields they may read or write. Finally, both the dynamically-inferred model (which tends to be accurate but incomplete) and the statically-identified dependence information (which tends to be conservative) guide a random test generator to create legal and behaviorally-diverse tests. Combining these three steps permits Palus to create good tests, and scales to large programs.

Palus is not the first technique that employs a dynamic execution trace to guide test generation. Several past research tools follow an approach similar to Palus' technique, but omit one or two of the three stages of Palus. Randoop [148] is a pure random test generation tool. Palulu [10] is a representative technique to combine dynamic analysis and random testing. Like Palus, it infers a call sequence model from a sample execution, and follows that model to create tests. However, compared to Palus's approach, the Palulu model lacks constraints for method arguments and has no static analysis phase to enrich the dynamically-inferred model with information about methods that are not covered by the sample execution. Finally, RecGen [40] uses a static-random approach. RecGen does not have a dynamic phase, and uses a static analysis to guide random test generation. Lacking guidance from an accurate dynamic analysis, RecGen may fail to create legal sequences for programs with constrained interfaces, and may miss many target states (as shown in our experiments [236]).

Another two alternative approaches to improve effectiveness of automated test generation are via direct heap manipulation and using capture and replay techniques. Korat [22] and TestEra [131] are two representative techniques that directly manipulate the heap to construct objects. They require users either to provide a `repOK()` predicate method or class invariants in the Alloy specification [89] language. In contrast, our Palus technique does not require a manually-written invariant to create inputs. Instead, Palus infers a model and uses static analysis to guide the random search towards legal and diverse sequences. On the other hand, the Object Capture-Based Automated Test (OCAT) approach [92] uses a capture and replay technique to save object instances from sample executions, and then reuse these captured object instances as argument values when creating sequences. Compared to

Palus, OCAT reuses the saved object instances (serialized on disk) and does not create a sequence to generate that object instance. That might be less intuitive for developers to understand where the object instance comes from. Besides, OCAT is designed as a regression test generation technique, and does not achieve the objective of bug finding (in either their methodology or experimental results). Test carving [48] and test factoring [167] also use capture and replay techniques, but have orthogonal focus than our work. They aim to capture the interactions between the target unit and its context and then create the scaffolding to replay just those interactions. The major focus of these two techniques is to generate small unit tests from system test cases and reduce average test suite, instead of creating new tests to increase coverage and find new bugs.

The theory structure used in Palus could be treated as a Java implementation of Parameterized Unit Test (PUT) [192]. There are also several recent works to create PUTs to improve structural coverage [190] or find bugs [192]. In contrast, Palus uses a theory as an oracle to check the intended program behavior and does not try to convert generated sequences into a PUT.

### *7.2.2 Testing Configurable Software*

Empirical studies show that configuration errors are pervasive, costly, and time-consuming to diagnose [87, 222]. To alleviate this problem, researchers have designed various software testing techniques to validate a configurable software system's behavior [63, 64, 70, 98, 112, 143, 158]. For example, Garvin et al. [63] show how to use historical data to avoid system failures during reconfiguration. Qu et al. [158] propose a combinatorial interaction testing technique to model and generate configuration samples for use in regression testing. Recently, Song et al. [177] present iTree to improve combinatorial interaction testing by discovering sets of configurations to test that are smaller but can achieve higher testing coverage.

Existing testing techniques are complementary to our ConfDiagnoser technique described in Chapter 4, though they have a rather different goal. These testing techniques aim to find new errors in a configurable software system earlier and reduce the burden of configuration management, while ConfDiagnoser aims to identify the root cause of a *revealed* configuration error.

### *7.2.3 Testing GUI Applications*

The only way for most software end-users to interact with a software application is through its Graphical User Interface (GUI). Automated GUI testing is a challenging task, since GUI applications have different characteristics from conventional software due to their event-driven nature. Various techniques are developed to automatically create tests for GUI applications [85, 224, 225]. For example, Guitar [224, 225] is a GUI testing framework for Java and Microsoft Windows applications. Yuan and Memon [225] generate event-sequence-based test cases for GUI programs using a structural event generation graph. Compared to the UI action sequence generation algorithm in FlowFixer (Section 6.2.4), existing techniques are model-based: they often derive graph models that approximate the possible sequences of events of the GUI [224, 225], and then use these models to derive representative test sets (i.e., UI action sequences). Such model-based techniques may not correctly account for interdependencies and interactions between different actions (an action could only become available after successfully performing another action) since the models are only approximations of real application behavior. In general, a test case directly derived from graph models can be infeasible, so effort has been put into repairing infeasible tests [134]. By contrast, FlowFixer incrementally builds each UI action sequence by using feedback obtained from executing the sequence as it is being constructed. This guides the search towards feasible sequences. On the other hand, existing model-based techniques are complementary to the randomized algorithm used in FlowFixer: model-based techniques explicitly try to maximize the model (or code) coverage of the whole GUI application, while a randomized algorithm has no such guarantee.

Michail and Xie [138] propose a tool-based approach to help users avoid bugs in GUI applications. Their approach monitors a user's actions in the background, and gives a warning as well as the opportunity to abort the action, when a user attempts an action that has led to problems in the past. Their work aims to *prevent* an existing bug from happening again. By contrast, the FlowFixer technique described in this dissertation aims to *repair* a broken workflow.

There is a large body of research on maintaining GUI code. Li and Wohlstadter [119] use dynamic information to show generated GUI widgets in a GUI editor and to help GUI editors map source code to GUI views when code changes. To help in understanding GUI test scripts, Fu et al. introduce an approach to infer the type of widgets and map them to code components [58]. Grechanik et al.

offer an approach to identify changes between two GUI applications and automatically maintain test scripts by comparing the successive GUI trees [67]. Recently, Wang et al. [205] propose an approach to automate presentation changes in dynamic web applications. Their approach helps a developer perform presentation changes in a dynamic web application in a way similar to performing presentation changes in static web pages. The proposed approach recommends UI-related code changes by mapping changes from the program output to its source code using dynamic analysis and check the safety of the changes through static analysis and dynamic analysis. Compared to the FlowFixer technique, existing techniques identify code fragment, test scripts, or GUI elements that are affected during GUI application evolution, but cannot repair broken workflows encountered by software end-users.

A number of approaches have been proposed to automatically repair obsolete GUI test scripts. For example, Huang et al. use genetic algorithms to repair infeasible GUI test cases and to generate new test cases [85]. Memon et al. propose a technique that repairs obsolete GUI tests by modeling GUIs with event-flow graphs [134], Nagarajan et al. propose an event-based profiling for refactoring GUI-based applications via changes in the GUI layout and removal of unused event handlers [142]. The work by Brett et al. [42] introduces a set of GUI refactoring types to help developers accommodate the changes both in GUI views and their corresponding controllers. Significantly different from FlowFixer, existing approaches focus on repairing *unusable* tests rather than *broken* workflows. Unlike a broken workflow, an unusable test is generated because the models for a GUI application are only approximations of the actual event flows, so that some events in a generated test may not be available for execution and cause the test to terminate prematurely. To repair an unusable test, existing techniques replace infeasible events in the test with alternative feasible events *without* preserving the actual testing logic and semantics. By contrast, FlowFixer fixes a broken workflow (caused by software evolution) by suggesting replacement actions. The repaired workflow preserves the "semantics" of the original workflow and completes the *same* task in the updated GUI application.

### *7.3 Software Error Diagnosis and Patching*

Four techniques described in this dissertation are related to research in the field of automated software debugging and error patching: FailureDoc explains why a test fails (Chapter 3), ConfDiag-

noser and ConfSuggester diagnose software configuration errors (Chapters 4 and 5), and FlowFixer helps end-users repair a broken workflow (Chapter 6). We next discuss closely related work to our techniques.

### *7.3.1 Automated Debugging Techniques*

**Dynamic Analysis Based Debugging**

Software developers have long struggled with understanding the causes of software bugs. To reduce the cost of identifying the root cause of a bug, many automated debugging techniques [34, 82, 102, 227, 229] have been developed.

Program slicing [196] is a well-known technique to determine which statements and inputs are relevant with a particular variable. The concept of program slicing was first defined by Mark Weiser, who introduced program slicing as a debugging aid and gave the first static slicing algorithm. After recognizing the need for accurate slicing, Korel and Laski proposed the idea of dynamic slicing [2], in which the data dependences exercising during a program execution are captured precisely and saved. A program slice [196] is a subset of the program execution that is relevant for a specific state or behavior. Slices are based on dependencies between statements: A statement $s_2$ depends on a statement $s_1$, if $s_1$ can influence the program state accessed by $s_2$. Starting from a statement, the transitive closure over all dependencies forms a program slice. In debugging, computing the backward slice for a failing statement returns all statements that could have influenced the failure. However, even though slicing can considerably reduce the number of relevant statements, the sets of relevant statements identified are still too large to be useful for debugging [240], and slicing-based debugging techniques are rarely used in practice.

Another line of automated debugging techniques is characterized by their experimental approach. Rather than assuming a set of executions, these techniques generate additional executions whose outcome guides the systematic isolation of failure causes. The first representative of these techniques is Delta debugging, an approach starting with only one failing and one passing run and isolating minimal failure-inducing differences in inputs [228], code changes [226], or program states [227]. More recently, predicate switching has been developed as an approach that flips branch outcomes [239] during execution in order to identify defect-related branches. As they operate on a potentially unlimited

number of executions, experimental approaches can narrow down failure causes with high precision. However, they manipulate executions in a way that may be unsound and may thus raise infeasibility issues. Such infeasibility may in turn compromise diagnoses and make it harder to understand the results when they involve state or branch manipulation. The FailureDoc technique described in Chapter 3 is an experimental approach, as it systematically replaces values in the original failed test to create mutated tests (and executions) and uses the outcomes of the mutated tests to isolate failure causes. FailureDoc therefore also assumes the presence of an oracle — that is, an automated means to distinguish passing from failing tests. This can be as simple as a single assertion or even the null oracle (no oracle at all) in case of errors detected by the runtime system. However, in contrast to techniques such as Delta debugging or predicate switching, the additional executions FailureDoc generates are always feasible.

An alternative family of debugging techniques aims to overcome the limitations of slicing-based approaches and experimental approaches by following a different philosophy. These techniques identify potentially faulty code by observing the characteristics of failing program executions, and often comparing them to the characteristics of passing executions. Tarantula is a representative automated debugging technique in this family [102]. Specifically, Tarantula utilizes test execution information: the pass/fail information about each test case, the entities that are executed by each test case (e.g., statements, branches, and methods), and the source code for the program under test. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. Based on the test execution information, Tarantula ranks the likelihood of each statement being buggy and produces a ranked list. Other approaches in this family differ from one another in the type of information they use to characterize executions – path profiles [30], synchronization coverage [151], statement coverage [9], change coverage [183], and predicate values [121] — and in the specific type of mining performed on such information.

Capture/replay-based techniques have been successfully used to support automated debugging tasks, as they are being combined with diagnostic features. Test factoring [167] and test carving [48] capture at the method level to extract unit tests specific to a task. The ReCrash tool [11] records executions by recording parts of the program state at each method entry – namely those objects that are reachable via direct references. When the program crashes, it thus allows the developer to observe

a run in several states before the actual crash. This is very efficient, but assumes that the stack trace actually contains the code (and state) that causes the failure.

One problem with the above debugging techniques is that they focus exclusively on trying to reduce the number of statements (or other program elements) developers need to examine when investigating a bug, under the assumption that examining a faulty statement in isolation is enough for a developer to detect and fix the corresponding bug. Unfortunately, it is not always true in practice. In fact, as pointed out by a recent study, even with the aid of automated debugging tools, it is still difficult for developers to actually determine the faulty nature of a statement by simply looking at it, without any additional information [152]. To overcome this limitation, FailureDoc takes a different perspective to help users diagnose a test failure. Instead of pinpointing the likely buggy statements, FailureDoc supports the developer with additional information about a failure by augmenting a failed test with debugging clues. As suggested by our user studies [238], FailureDoc helps developers understand the failure cause faster.

Dialog-oriented debugging approaches also provide explicit support for failure understanding. Pothier et al. [154] present a portable Trace-Oriented Debugger for Java which uses efficient instrumentation techniques for event generation and a scalable storage system for completeness and efficient querying. Their technique permits developers to query the recorded execution trace for narrowing down possible failure causes. Whyline is a tool from Ko and colleagues [113] that allows developers to ask questions about the visual output of a program based on a recorded execution. The tool by Hao and colleagues [77] suggests breakpoints to the developer for an interactive localization of the defect. By contrast, FailureDoc is significantly different from these approaches. First, FailureDoc pinpoints relevant code part in a failed *test*, which is often a starting point for debugging; while existing approaches localize suspicious statements in the source code. Second, FailureDoc compares a failing execution trace with a set of passing execution traces to generate useful properties and explain why a test fails, while previous work only pinpoints the relevant source code and does not generalize the observations into high-level explanation.

**Static Analysis Based Debugging**

There is also a large group of static debugging techniques. The philosophy of static debugging is to first construct a correct model by specifying certain rules, and then statically analyze the

target program to see if these rules are strictly followed. For example, FindBugs [83] uses static analysis to locate a set of predefined, common bug patterns in code (e.g., equal objects must have equal hashcodes). LCLint [51] uses annotations to represent assumptions about function interface, variables, and type explicit. Constraints derived from these annotations are checked at compile time. Any violations are considered as potential errors. CQual [56] proposes a technique in which users annotate their programs with flow sensitive type qualifiers. The correctness of the programs can be checked by inference. Prefix [26] presents an important methodology of using static program analysis to detect memory related program errors. The idea is to symbolically execute the functions, modeling the memory and reporting any inconsistencies. SLAM [16] uses techniques from program analysis, model checking, and automated deduction to check whether a C program follows certain rules in using an API. Blast [79] is a similar technique which performs model checking on the safety properties of C programs. In [90], the code of a procedure is modeled as relational formulas, which are conjoined with the negation of the procedures specification. A constraint solver is used to either verify the procedure or generate counter-examples. In [218], Xie and Aiken translate C programs into boolean formulas such that boolean satisfiability solvers can be used to efficiently check any violations to certain specified properties. In [130], Manevich et al. propose using post-mortem static analysis to locate faults. The idea is that after knowing the type of failure and the program location where the failure happened, static analysis such as pointer analysis can be performed more effectively such that the flow of a value can be more accurately pinpointed.

Compared to the dynamic approaches in this dissertation, static techniques usually require developers to write annotations or specifications. A considerable number of false positives are usually incurred by the conservative nature of underlying static analysis such as pointer analysis. Additionally, static analysis may fail to report true but computationally expensive program properties. By contrast, dynamic analysis, which runs the program, examines the executions, and reports properties over those executions, does not suffer these drawbacks and so complements static analysis [50].

**Debugging Evolving Software.**

Validation of evolving software is an important problem, since any large software moves from one version to another. Among the established efforts in this direction are the works on regression error localization which focus on identifying the failure-inducing changes of a regression failure.

Existing work on regression error localization can be classified into two major categories: (1)

statically analyzing the source code to isolate a minimal subset of changes that introduce the regression failure [8, 161, 226], and (2) dynamically analyzing the failing program execution [182, 241], or comparing the failing program execution with one chosen program execution which does not manifest the observable error in question [162, 185, 227].

In the first category, program differencing methods and Delta debugging are two representative techniques. Program differencing methods [8, 81, 221] try to identify changes across two program versions. Indeed, this can be the first step towards detecting errors introduced by program changes – identifying the changes themselves. The work on change impact analysis [118, 161] is often built on such program differencing methods where the analysis identifies not only the changes between two program versions, but also which tests are affected by which changes. However, the work based on program differencing tries to statically identify possible software changes, but cannot find the root cause of a given software regression error. Further, the output of a program differencing tool often requires human inspection to identify likely buggy changes. Delta debugging techniques aim to find a minimal subset of changes that still makes the test fail. The work of [226] studies debugging of evolving programs and proposes to identify failure-inducing changes. Zhang et al. [234] combines Delta debugging and program differencing to improve efficiency. However, all these techniques focus on only a minimal set of changes that make tests fail, because they implicitly assume that the cause of the regression error is contained in that set. In other words, such techniques are restricted to only reporting the changes as error causes. Errors present in the old version which get manifested due to changes cannot be explained using such an approach. For example, a configuration option which is mistakenly set to an incorrect value in the old version but never used is indicative of such a situation. Such configuration error may only be observed in the new version when the configuration option value is used. A good diagnosis report should pinpoint the fact that this configuration option is mistakenly set, not just the changed code where it affects. Further, for many configuration errors such as the example discussed in Section 4.2, the error-revealing inputs are often minimized. This makes program differencing based methods and Delta debugging ineffective.

In the second category, given two program versions and an observable error from a failing execution, dynamic analysis based approaches try to find the root cause of the observable error by either analyzing the observed failing execution or comparing the failing execution trace with some chosen correct executions. The family of spectrum-based [1, 72, 107, 156, 169] bug localization

techniques often use the longest common string (LCS) algorithm to compare two execution traces (a passing one and a failing one), and consider the different parts more likely to contain the failure cause. Combining with Delta debugging, Gupta et al. [73] propose a hybrid approach to use Delta debugging to identify a minimal failure-inducing input, use this input to compute a forward dynamic slice, and then intersect the statements in this forward dynamic slice with the statements in the backward dynamic slice of the erroneous output to compute a failure-inducing chop. Their experiments indicate that failure-inducing chops are more likely to contain the root cause of a regression error than other program parts. Recently, Hoffman et al. [80] present a dynamic analysis to localize regression failures on top of a semantic view of program executions. Their view definitions reflect program abstractions and aspects. Views are not simply projections of execution traces, but are linked to each other to capture semantic interactions among abstractions at different levels of granularity. Compared to spectrum-based algorithms, Hoffman's approach generates more concise and accurate bug diagnosis reports, since it can identify syntactically-different but semantically-equivalent execution trace components and avoid reporting those parts in the result. Qi et al. [157] propose a symbolic execution-based approach for debugging evolving software. Their approach, called Darwin, takes a different perspective to address this problem. Darwin first uses symbolic execution to synthesize new inputs that differ marginally from the failing input in their control flow behavior, then compares the execution traces of the failing input and the new inputs to obtain critical clues to the root cause of the failure.

Our FailureDoc technique bears some resemblance to the work that compares a failing program execution with a set of chosen correct program executions. However, FailureDoc only takes as input a single failing execution. It uses value replacement [97, 238] to create a number of passing executions for comparison. Further, FailureDoc generalizes the difference between a passing execution and some failing executions for creating documentation, while existing techniques lack this phase.

Our ConfDiagnoser technique also compares a failing execution trace with a number of selected passing execution traces to diagnose a configuration error. Differing from existing techniques, ConfDiagnoser correlates a configuration option-affected predicate's true ratio and execution frequency with the observed behaviors, rather than just its evaluation result (i.e., either true or false of a predicate). Further, similar techniques like CBI [121] only identify likely buggy statements as their final output, while ConfDiagnoser identifies behaviorally-deviated program predicates and links their

undesired behaviors to specific configuration options.

### *7.3.2 Diagnosis of Configuration Errors*

Software configuration error diagnosis is recognized as an important research problem. Many techniques have been developed to assist developers and system administrators in debugging software configuration errors [12–14, 109, 159, 203, 209].

Program slicing [82] and taint analysis [34] are two well-known techniques to determine which statements and inputs could affect a particular variable. Despite their effectiveness in diagnosing a crashing configuration error by performing backward reachability analysis from the crashing point [14, 159], these two techniques cannot be directly applied to diagnose a non-crashing error.

Chronus [209] relies on a user-provided testing oracle to check the behavior of the system, and uses virtual machine checkpoint and binary search to find the point in time where the program behavior switched from correct to incorrect. AutoBash [184] fixes a misconfiguration by using OS-level speculation execution to try possible configurations, examine their effects, and roll them back when necessary. PeerPressure [203] uses statistical methods to compare configuration states in the Windows registry on different machines. When a registry entry value on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error. More recently, ConfAid [14] uses dynamic taint analysis to diagnose configuration problems by monitoring causality within the program binary as it executes. ConfAnalyzer [159] uses dynamic information flow analysis to precompute possible configuration error diagnoses for every possible crashing point in a program. X-Ray [12] uses dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause, then summarizes the overall cost of each potential root cause by summing the per-block cost multiplied by the cause-specific likelihood over all basic blocks.

Our ConfDiagnoser and ConfSuggester techniques are significantly different from the existing approaches. First, most existing approaches focus exclusively on configuration errors that lead to software crash or assertion failures [13, 14, 159, 209], for which they can leverage valuable crashing information such as stack traces. By contrast, our techniques can diagnose both *crashing* and *non-crashing* errors. Second, several existing approaches [14, 209] assume the existence of a testing oracle

that can check whether the software functions correctly. However, such oracles are often absent in practice. Writing them may require a substantial amount of time and effort that a typical software user would not prefer, and should not be expected, to invest. By contrast, our techniques eliminate this assumption. Third, approaches like PeerPressure [203] benefit from the known schema of the Windows registry, but cannot detect configuration errors that lie outside the registry. Approaches like X-Ray [12] focuses on diagnosing performance-related configuration problems, and it is unclear whether it can be extended for diagnosing other configuration errors. Our technique of analyzing the affected predicate behavior is more general for configurable software. Fourth, our techniques do not need any OS-level support. It requires no alterations to the JVM or standard library. This distinguishes our work from competing techniques such as OS-level configuration error troubleshooting [184, 209].

### *7.3.3 Software Failure Explanation*

Although many automated testing and debugging techniques and tools [32, 40, 146, 148, 231] have been developed to identify the root causes of software failures, only a few approaches explicitly try to help developers understand a failure. Tarantula, for instance, visualizes test information and highlights suspicious code [102], but provides no further explanation. The cause-effect chains of Zeller [227] explain a failure in terms of how variable values cause each other through a program execution. Delta debugging [228] is a general technique to isolate failure-inducing inputs. It has the potential to isolate suspicious statements from a failed test, but cannot explain *why* an isolated statement is suspicious. BUGEX [166] correlates runtime facts with the revealed bug, including branches taken (implying the statements executed) or variables with specific values. However, none of these existing techniques can explain a failed test. With the wider adoption of automated testing tools, quickly understanding failed tests becomes a demanding requirement. The FailureDoc technique described in this dissertation addresses this problem (Chapter 3). Fundamentally different from the existing work, FailureDoc infers descriptive code comments as debugging clues for a given failed test by changing variable values of the test and associating these values with the execution outcomes. As shown in our user studies [238], the generated comments explicitly inform developers (or testers) which parts of the test are relevant to the failure before they start bug-fixing.

Recent statistical techniques have been applied feature selection, clustering, and multivariate

visualization techniques to the task of identifying and classifying software failures [46, 121, 123, 242]. In particular, the CBI project [121] analyzes execution traces collected from deployed software to isolate software failure causes. Specifically, the program is instrumented to collect information from certain runtime values and this information is passed on to a statistical engine to compute likely buggy predicates. Compared to the CBI project, FailureDoc has several differences and a rather different goal. First, instead of having many collected execution traces, FailureDoc is given only one failed test (i.e., a single failing execution trace). It needs to construct extra execution traces before applying a statistical algorithm. FailureDoc addresses this problem by using value replacement to construct slightly mutated tests and then executing them to obtain extra execution traces. Second, the CBI project uses the value (true/false) of an instrumented predicate as the feature vector, while FailureDoc uses a finer granularity: it observes and records multiple computed values of related expressions. Third, FailureDoc does not track the usually long execution trace across the whole program as CBI does. FailureDoc's static analysis and runtime monitoring is intra-procedural, permitting lower overhead. Fourth, CBI and FailureDoc have different goals. CBI uses statistical algorithms to identify likely buggy predicates as final output, while FailureDoc identifies a set of suspicious statements and their failure-correcting objects for documentation inference.

There are also some techniques proposed in the model checking community for explaining a counterexample [19, 69, 117]. Compared to FailureDoc, explaining a counterexample has a rather different goal. Representative work like [69] is similar to Delta debugging [228]. It takes an error trace produced by a model checker, and computes a minimal transformation between error and correct traces by conducting a modified binary search over program states. In contrast, FailureDoc takes as input a failed test, creates additional execution traces, and generalizes failure-correcting edits as documentation to explain its failure.

There has been some limited work on automated documentation inference for source code. Semi-automated approaches like [164, 165] either determine un-commented code segments and prompt developers to enter comments, or generate comments from user-provided high level abstractions. Some techniques use static analysis to generate comments for exceptions [24], API function cross-references [52], software changes [25, 110], and descriptive summary comments for methods [180, 181]. However, none of the previous work can generate documentation to explain a failed test, due to the different abstractions they employ. In practice, a considerable part of the development cycle

is spent in debugging. During debugging, developers or testers often need to inspect a failed test, understand its failure cause, and localize the revealed bug. The difficulties raised in understanding a failed test motivate the FailureDoc work.

### *7.3.4 Patching Software Errors*

Research in automatic error recovery and repair has ranged from using formal specifications and repair templates to detect and patch errors [45]; to automatically inserting code to handle overflow errors using genetic programming [116], to enforcing software behaviors at runtime [27, 128, 153]. Several other recent but less mature proposals in this area, e.g., [20] suggests that we can expect rapid progress over the next several years.

In the category of *enforcing software behaviors at runtime*, the work by Demsky et al. [45] combines runtime monitoring and formal specifications to detect and repair inconsistent data structures. ClearView [153] enforces learned invariants to eliminate errors and vulnerabilities. Specifically, ClearView learns invariants over registers and memory locations, detects critical invariants that are violated, then generates and installs patches that eliminate the potential errors by modifying the program state to enforce the invariants. Following a similar approach, Jolt [27] monitors an application's progress, records the program state at the start of each loop iteration, then reports to the users about potential infinite loops if two consecutive loop iterations produce the same state. By enforcing program control to a statement following the loop, Jolt helps the application recover from infinite loops. More recently, input rectification [128] has been proposed as a technique to avoid software errors by automatically converting potentially malicious inputs into typical inputs that the program is highly likely to process correctly. Like ClearView, it learns a set of constraints characterizing typical inputs that an application is highly likely to process correctly, and then uses the learned constraints to enforce the software behaviors when processing a likely malicious input.

Predicate switching [239] is a general method to automated debugging. Given a failed execution, by repeatedly and forcibly switching a predicate's outcome at runtime and altering the control flow, predicate switching modifies the program state and brings the failed execution to a successful completion (i.e., program produces the desired output). By examining the switched predicates, the cause of the bug can often be identified.

Although repairing broken workflows can be viewed as a special case of automated error patching, the FlowFixer technique described in Chapter 6 is significantly different from existing error patching techniques. Existing techniques usually search for good patches using specifications or test oracles as the criterion, and are concerned with bugs in the program that result in erroneous program states. By contrast, FlowFixer does not try to fix the program itself. Instead, it repairs a broken workflow caused by UI changes that are not related to any erroneous program states.

# Chapter 8

# FUTURE WORK

While this dissertation has demonstrated some of the potential of program analyses for improving automated software testing and end-user error diagnosis, many opportunities for extending the work in different ways remain. This chapter discusses several potential directions.

## *8.1 Developing User Interfaces for Program Analysis Tools*

A good user interface can improve a program analysis tool's usability. This is particularly true for non-expert end-users, who may have little or no programming knowledge. We plan to build user interfaces for several tools developed in this dissertation. For example, the current implementation of FlowFixer outputs the suggested fixes as plain text, such as: "Click menu item $Files \rightarrow New\ Crossword$". Although useful, such plain text requires an additional effort from users to localize the desired UI element on the software UI. A graphical user interface can help visualize the analysis result, such as by moving a user's mouse to the $Files \rightarrow New\ Crossword$ menu item or highlighting the specific UI parts that a user should explore. The need to build a good user interface also applies to tools designed for software developers. For example, building a user interface for Palus can permit developers to inspect the constructed call sequence graph conveniently, understand why a test is (or is not) generated, and identify whether the provided usage examples or test cases are adequate or not. Palus's UI could also cluster tests generated by extending the same transition edge together, and overcome the potential difficulty of finding a relevant test in Palus's output. A user interface for FailureDoc would be useful too. Developers can select only that part of the program or test that is of interest, specify certain variables to include or omit, and filter away some redundant code comments. A developer may also add project-specific or domain-specific test inputs or templates to help FailureDoc derive more useful properties.

### *8.2 Conducting User Studies to Understand Analysis Usability*

Besides applying each developed program analysis to more and larger programs, understanding and investigating tool usability to real-world users is an important aspect of future work.

We plan to conduct user studies for each developed technique in the future to answer the following important research questions: How steep is the learning curve for use of an automated analysis tool? How easy is the process of interpreting an analysis' output? Are automated program analysis tools actually useful in creating good test suites, understanding software failures, and troubleshooting end-user errors? If so, how much time and human effort can be saved? What programming tasks or user activities are automated tools most helpful for, and which tasks are not? Are automated tools more useful to certain users than to others? How can program-analysis-based tools complement other available methods (e.g., searching relevant information on Google) for error diagnosis? Answers to these research questions will help us better understand the usefulness of the developed techniques, discover existing design flaws, identify new improvement space, and refine the future implementation.

### *8.3 Improving Non-Functional Software Properties*

Besides checking the functional correctness of a software system, another aspect of my future research agenda is making program analysis techniques and tools verify and improve non-functional software properties, so that software developers can use these techniques to build more reliable, efficient, secure, and energy-efficient software systems.

We first plan to apply the ideas of Palus and FailureDoc to new problem domains, such as performance and energy testing on mobile devices [211] and security testing of safety-critical systems [149, 176, 214, 215]. The power of Palus [236] and FailureDoc [238] comes from combining static and dynamic program analyses with observation generalization. We believe this hybrid analysis pipeline is applicable to other software development problems, such as finding performance bottlenecks, energy hotspots, and security vulnerabilities. For example, one could use dynamic analysis and observation generalization to learn a "secure behavioral model" of the tested software, and then use the learned model to guide static analyses (such as symbolic execution or constraint solving techniques) to generate inputs that cover the likely insecure program parts to expose security vulnerabilities. Another direction is to improve program-analysis-based techniques with patterns

mined from program code or version control histories [188, 220, 244]. For example, a buffer overflow attack might often be fixed by inserting a bound check. Designing tools leveraging such patterns may provide developers richer contextual information in error diagnosis and elimination. One challenge is mining succinct and general patterns without overfitting specific subject programs.

### *8.4 Classifying Software Errors*

When a software system exhibits some erroneous behavior, the user must understand its root cause before taking any action to fix it. Despite the recent advance in automated program debugging and error diagnosis, few techniques have been developed to inform software users of the nature of a software error. Is the error because of a source code bug? Is the error because the software is misconfigured? Or, is the error because a wrong input was provided to the software system? Answers to the above questions are critical and can significantly affect a software developer or user's decision in choosing the proper diagnosis technique. However, existing automated debugging or error diagnosis techniques do not explicitly address this problem; rather, they often assume the error is caused by a certain type of root causes. For example, Delta debugging [226] assumes the software error is caused by an illegal input, our ConfDiagnoser and ConfSuggester techniques assume the undesired behavior is caused by a configuration error, and some other state-of-the-art automated debugging techniques [101, 234] assume the error is caused by a source code bug.

As a first step, we plan to enhance both ConfDiagnoser and ConfSuggester techniques by developing techniques that automatically distinguish source code bugs from configuration errors, when a software system exhibits some undesired behavior. Such techniques can help formulate guidance regarding when the user should give up on our tools and assume the error is not related to a configuration option. More broadly, we plan to develop a classification system to distinguish different types of software errors and their root causes. Such a system could be built on recent progress on failure clustering (that clusters similar software errors together) [29, 46] and machine learning (that extracts error features, constructs a classification model, and predicts the type of a new error) [74].

### *8.5 Recovering from Software Failures*

One challenge for achieving software reliability is to deal with the size, the complexity, and the heterogeneity of modern software systems. As part of our future research agenda, we are particularly interested in designing analyses and runtimes that can automatically bypass or even patch software failures. Software in general suffers a notorious lack of memory: a program that crashes once may crash again, with no apparent knowledge of its previous failure. We plan to develop techniques that allow software systems to learn from their mistakes and adapt to their failures, much like just-in-time compilers adapt to new information about program usage to improve performance. The approach could build on recent progress on software speculative analysis (that predicts the future state of a software system) [23] and genetic programming techniques (that generate bug fixes to patch a software failure) [116].

We next describe a concrete idea of how to patch software configuration errors. We propose an automated technique called ConfErrFixer as an extension of ConfDiagnoser. ConfErrFixer works directly on the binary code of the program-to-fix, and does not require hard-to-write formal specifications, program annotations, or special coding practices. When patching a configuration error, it takes as input a program, an error-revealing input and configuration, a testing oracle that decides whether the patched program functions correctly, and the diagnosis report produced by ConfDiagnoser (i.e., a list of suspicious configuration options with the identified behaviorally-deviated predicates as shown in Figure 4.1). ConfErrFixer repeatedly *enforces* the outcome of each behaviorally-deviated predicate at runtime until the program produces the desired output. The hypothesis of ConfErrFixer is that in an undesired execution trace, there are often a small number of *critical predicates* whose runtime outcomes, if changed to similar outcomes as observed in the correct execution traces, would bring the program to produce the desired output. Based on this hypothesis, ConfErrFixer systematically searches for such critical predicates, and generates a patch for each of the critical predicates, specifically by generating a snippet of Java bytecode that checks and enforces the outcome of a critical predicate.

We use the Randoop code excerpt and ConfDiagnoser's diagnosis report in Figure 4.2 as an illustrating example. As indicated by ConfDiagnoser's diagnosis report in Figure 4.1, a predicate named "`newSequence.size() > GenInputsAbstract.maxsize`" behaves dramatically dif-

ferently between the recorded undesired execution and the correct executions. ConfErrFixer selects this predicate as a critical predicate candidate, and generates a patch that enforces its runtime behavior as observed in the correct executions (i.e., 32.5% of the time this predicate evaluates to true). After the enforcement, the configuration option `maxsize`'s pre-defined, incorrect value no longer takes effect; and some generated sequences that exceed the pre-defined limit (including sequences that the user wishes to remain) are included in the output. Hence, ConfErrFixer identifies the predicate "`newSequence.size() > GenInputsAbstract.maxsize`" as a critical predicate, and patches the exposed configuration error by using the generated code snippet.

To make ConfErrFixer feasible, we plan to restrict ConfErrFixer's search for critical predicates in ConfDiagnoser's diagnosis report. Since predicates in the report are the most behaviorally-deviated, enforcing their behaviors is more likely to produce desired output. ConfErrFixer considers each predicate in the diagnosis report as a critical predicate candidate. For a critical predicate candidate, ConfErrFixer uses its ratio of true outcomes observed in the correct execution traces to replace its actual evaluation outcome, and thus *forcibly* alters the program's control flow. More precisely, when evaluating a critical predicate candidate, ConfErrFixer employs a random boolean generator that produces true with the same ratio of true outcomes of the predicate in the correct execution traces, and then uses the generated boolean value to replace the predicate's actual evaluation outcome.

### *8.6 Analyzing Non-Executable Artifacts*

Source code (including its execution trace) is not the only artifact of the software development process. Other artifacts include design documentations, API documentations, user manuals, change logs, bug reports, external resources (e.g., web services), code comments, code reviews, change requests, emails, and the like. These artifacts are prevalent throughout the lifetime of a software system not just during its development phase. Analyzing them could potentially offer software practitioners (not just developers) up-to-date and pertinent information to support their daily decision-making processes. For example, some recent work on identifying duplicate bug reports [204] and analyzing natural language documentations [149, 150, 213] suggest that evidence from analyzing executable and non-executable artifacts are often strongly intermixed and evidence from one side can help resolve the uncertainty of the other. This principle is also applicable to many problems

described in this dissertation. For example, when diagnosing a configuration error, an automated tool like ConfDiagnoser may use natural language processing techniques to parse the error message (in plain text form) and then use the parsed information to guide the search of suspicious configuration options. For instance, if an error message contains a `ClassNotFoundException`, a diagnosis tool may first search and report classpath-related configuration options.

### *8.7 Preventing and Resolving End-user Errors*

As the complexity of modern software systems grows, end-users are more likely to make mistakes in using complex software systems. An end-user error can cause serious consequences. For example, a bad configuration setting in a browser or a firewall client can threaten user security and privacy. One aspect of our future research agenda is to empower software end-users with automated tools to prevent and resolve errors.

This dissertation has provided three program-analysis-based techniques to assist end-users in fixing functional configuration errors (Chapters 4 and 5) and GUI workflow errors (Chapter 6). In the future, we plan to develop techniques to diagnose and fix non-functional configuration errors related to end-user security and privacy. We also plan to develop techniques to automatically perform complex configuration tasks for software end-users, taking into account their use-case scenarios and preferences, and asking the user only for information that cannot be automatically inferred. For this, we plan to use a history-based approach and machine learning algorithms (such as collaborative filtering) to recommend configuration settings. Another direction is designing tools to help end-users proactively discover, understand, and adapt to software features, *before* they encounter a problem and introduce an error or a potential vulnerability. We also want to evaluate each developed tool with real end-users.

In the long term, we envision that end-users should have automated personal assistants powered by program analysis techniques. We are interested in developing techniques that can understand end-users' problems (possibly by natural language) and automatically troubleshoot them. These techniques could build on recent progress on lightweight, privacy-aware program monitoring [122], intelligent user interfaces, and automatic failure triage and removal — such as search-based techniques to find relevant solutions or patches on the web, or recommendation techniques to suggest feasible

workarounds.

### *8.8 Automating End-user Programming*

Many software end-users, such as research scientists and business analysts, are not professional programmers, but they often need to write simple programs (or scripts) to fulfill their domain-specific tasks. In our previous research [237] (not included in this dissertation), we showed how programming-by-example techniques help end-users write correct SQL queries. Our future goal in this direction is to empower end-users, with little or no programming knowledge, to write correct programs with minimal effort for a variety of systems such as spreadsheets, databases, mobile devices, robots, and intelligent tutoring systems [71, 78, 115, 175]. We plan to build automated or semi-automated tools empowered by program analysis techniques to achieve this goal. In the future, we plan to (1) empirically study common programming errors end-users make and understand the major barriers that they would face in practice; (2) design easy-to-use program analysis tools to identify, eliminate, and prevent such common errors; and (3) build automated program synthesis techniques to further reduce the barrier of end-user programming.

# Chapter 9

# CONCLUSION

This dissertation presents five program analysis techniques for improving automated software testing and end-user error diagnosis. The first technique, called Palus, combines dynamic model inference, static analysis, and random testing to automatically generate unit tests for software systems with constrained interfaces. Experimental results show that Palus is able to generate legal and behaviorally-diverse unit tests, achieving higher code coverage and detecting more bugs than tests generated by existing techniques. The second technique, called FailureDoc, assists developers in interpreting the test results. FailureDoc augments a failed test with contextually-relevant debugging clues: code comments that provide useful facts about the failure, and help developers understand why a test fails. Experimental results show that FailureDoc can infer meaningful code comments for most test failures, and the inferred code comments are useful for developers to understand the failure causes. The third technique, called ConfDiagnoser, assists software users in troubleshooting configuration errors. It uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options whose values should be changed. The fourth technique, called ConfSuggester, extends ConfDiagnoser to support configuration error diagnosis for evolving software systems. Experimental results show that both ConfDiagnoser and ConfSuggester are accurate and efficient. The fifth technique, called FlowFixer, helps end-users repair UI workflow errors. Given a new software version's changed UI, in which a user's desired action is not possible, FlowFixer informs users of replacement UI actions to fix a broken workflow. Experimental results show that FlowFixer can successfully repair a large portion of broken workflows, significantly outperforming existing techniques.

All techniques have been implemented, evaluated on real-world software systems, and compared to existing or alternative approaches.

## *9.1 Lessons Learned*

In this dissertation, we have developed new program analysis techniques and tools to improve the effectiveness of automated software testing and end-user error diagnosis. Our research is motivated by the need to improve the reliability of modern software and the need to perform end-user error diagnosis. Our initial results have provided concrete evidence to demonstrate the following principal thesis statement: automated software testing and end-user error diagnosis can be performed with minimal manual input and without formal specifications, by inferring software models and properties from execution traces and correlating software behavioral differences (captured by the inferred models and properties) to the root causes.

We summarize a set of lessons learned in this research. We focus on their successful application in this dissertation. Each of these lessons is a general observation and therefore previous works have likely demonstrated similar lessons. We still find it useful to list them here together as an explicit cohesive list of the high-level perspectives we gained from this dissertation.

**Integrate static and dynamic analyses.**

Static analysis examines program code and reasons over all possible behaviors that might arise at run time, while dynamic analysis operates by executing a program and observing the executions. Their mutual strengths can enhance one another. When designing the Palus technique, we hoped to extract the call sequence model statically from the source code in the beginning. However, several limitations of static analysis caused us to rethink this initial design. Specifically, extracting the call sequence model from source code requires static reasoning about object reference manipulation information, but precise modeling of heap state and pointers is still beyond the state of the art. By contrast, the dynamic analysis used in the current Palus design overcomes this limitation. It can detect context-dependent properties (e.g., method-call sequences) that may easily confuse static analyses. On the other hand, static analyses are complementary to dynamic approaches: they can reason over software behaviors that are missed by a dynamic analysis tool. Combining the best properties of static and dynamic analyses may be a promising way to solve many difficult problems.

**Use code analysis techniques to address non-source-code analysis problems.**

Configuration error diagnosis and GUI workflow repair are not typical software engineering problems. Most existing approaches to address these two problems treat the software system as a black box and thus primarily rely on some human-designed heuristics to reason about the error root causes as well as the fixes. It was not obvious to employ code analysis techniques to address these two problems in the beginning of the research. Our work on ConfDiagnoser and FlowFixer shows that source code analysis can enhance existing non-code-analysis techniques by resolving the uncertainty that cannot be resolved without analyzing the source code, such as distinguishing UI actions that look similar but result in different consequences. This experience suggests that source code analysis can be used as a powerful weapon to complement non-code-analysis techniques (or heuristics) to better understand certain program behaviors.

**Analyze software execution traces to obtain useful properties.**

Analyzing software execution traces is an effective way to obtain useful properties. It has two major advantages. First, software execution traces are pervasive and easy-to-obtain: developers often instrument key locations in the code to gain insight into the state of a program, the execution sequence, and the presence or absence of certain events. In practice, production systems at companies like Google and Microsoft are instrumented to generate billions of execution traces each day [220]. These traces are stored for weeks to help diagnose bugs and improve software quality. Second, compared to static source code analysis, execution traces are quite specific: they are obtained by executing a program with real inputs, they are precise and have no approximations, and there is little or no uncertainty in what control flow paths are taken, what values were computed, how much memory was consumed, how long the program takes to execute, or other quantities of interest. Thus, when designing a new automated testing or error diagnosis technique that tries to "understand" software code, execution traces become a useful source of information. This dissertation makes extensive use of software execution traces. All the developed techniques analyze software execution traces to infer, summarize, and generalize useful models and properties. Specifically, Palus analyzes method-call sequences in a correct execution trace; FailureDoc analyzes object values in multiple correct and incorrect execution traces; ConfDiagnoser and ConfSuggester analyze predicate execution behaviors in multiple correct and incorrect execution traces; and FlowFixer analyzes method invocations in

two different execution traces.

**Choose the right abstraction.**

A central task in designing a program analysis technique is to find a program abstraction that keeps just enough information and can be efficiently computed. The five program analysis techniques described in this dissertation use different program abstractions based on the nature of the targeted problems. For example, ConfDiagnoser and ConfSuggester use program predicates as the program abstraction, based on the key observation that control flow often propagates the majority of configuration-related effects and determines a program's execution path, while the value of a specific expression may be largely input-dependent. Experimental results show that choosing other abstractions, such as monitoring statement-level coverage or method-level invariants, yields less accurate results.

**Think about usefulness first.**

All program analysis techniques described in this dissertation are based on dynamic analyses to reason about program behaviors. The analysis results are not sound nor complete. However, in our evaluation and user study, we found unsoundness and incompleteness are of minor problems. The goal of our analyses is to improve software reliability (via automated testing), understand software failures, and diagnose software errors. While our analyses are not complete or contain some false positive, they are still useful. Software developers and users can always inspect a program analysis tool's output and ignore results that do not make sense to them. For example, Palus cannot guarantee that every test it generates is legal. But this has little impact to developers, since developers can always run the whole generated test suite and remove illegal tests that immediately fail.

**Contextually-relevant information is useful in debugging.**

Most existing automated debugging techniques assume perfect bug understanding, that is, simply examining a faulty statement in isolation is enough for a developer to detect, understand, and fix the corresponding bug. Unfortunately, perfect bug understanding rarely occurs in practice, as developers need additional information to understand software failures and correct bugs. Understanding the root cause of a failure typically involves complex activities, such

as navigating program dependencies and rerunning the program with different inputs. The FailureDoc technique described in this dissertation takes an initial step to providing test execution context as debugging clues. Our user study and the feedback from developers confirmed the usefulness of such richer contextual information.

**Filtering away irrelevant information is crucial in failure explanation.**

How to derive generalizable properties from a limited set of executions is a central problem for many dynamic analysis tools. The initial design of FailureDoc used a set of predefined templates to aggregate all matched properties and displayed them to the users. However, such a straightforward approach suffers from several limitations. First, it is difficult or sometimes even impossible to display a high volume of runtime information to software developers. Second, the problem was exacerbated when too much irrelevant information was reported, and such irrelevant information distracted user attentions and decreased a tool's usability. Filtering away (or hiding) likely unuseful information is as important as deriving useful information in the design of a failure explanation tool.

**User demonstration is a convenient way for human-tool cooperation.**

To increase the value of tool automation, cooperation between human and tools should be supported. A program analysis tool should provide convenient ways for its users (including developers or end-users) to express their goals, and a tool should also provide effective ways to communicate the analysis results to their users — enabling a feedback loop between human and tools to accomplish the goals. Designing an effective way for human-tool cooperation is particularly challenging for ordinary end-users with little or no programming knowledge. It is unrealistic to assume that all tool users can communicate with a tool using code fragments, partial specifications, or annotations of any form. In the design of ConfDiagnoser, ConfSuggester, and FlowFixer, we chose user demonstration as a convenient interaction way, since it is one of the easiest ways for end-users to describe their tasks.

**Show the other side of the coin.**

The FailureDoc failure explanation technique is driven by experiments — test runs that narrow down failure causes by systematically confirming or excluding individual observations. It

creates the opposite behavior of a software failure by repeatedly mutating a failed test. The created correct behaviors are then compared with the undesired behavior (i.e., the failure) to infer explanations that are strongly correlated with the failure. Besides explaining a software failure, knowing the correct software behaviors is also useful for diagnosing end-user errors. The ConfDiagnoser, ConfSuggester, and FlowFixer techniques follow this approach to isolate root causes from an undesired execution trace .

**Be simple, but not too simple.**

The basic idea of four techniques (FailureDoc, ConfDiagnoser, ConfSuggester, and FlowFixer) described in this dissertation is fairly simple: comparing a failing execution trace (or a set of failing execution traces) with a set of correct execution traces to identify the different parts, and then correlating such execution differences to a certain type of error root causes (i.e., an interested variable in FailureDoc, a suspicious configuration option in ConfDiagnoser or ConfSuggester, and a replacement UI action in FlowFixer). However, being simple alone is not enough; a considerable amount of design and engineering efforts have been spent to make simple approaches work, including carefully choosing program abstractions to keep needed information, evaluating tradeoffs between different designs, filtering away redundant or irrelevant information, and optimizing implementations to scale to realistic programs. These efforts are key to the success of each technique in this dissertation.

# BIBLIOGRAPHY


[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAIC PART-Mutation*, 2007.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, 2002.

[4] Apache Commons Collections. `http://commons.apache.org/collections/`.

[5] Apache Commons Math. `http://commons.apache.org/math/`.

[6] Apache Commons Primitives. `http://commons.apache.org/primitives/`.

[7] Apache Server. `http://httpd.apache.org/`.

[8] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *ASE*, 2004.

[9] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *ICSE*, 2010.

[10] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS*, 2006.

[11] Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP*, 2008.

[12] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[13] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, 2008.

[14] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[15] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA*, 1996.

[16] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.

[17] BCEL project page. `http://jakarta.apache.org/bcel/`.

[18] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, 2011.

[19] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. In *CAV*, 2009.

[20] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic web sites. In *WWW*, 2002.

[21] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[22] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA*, 2002.

[23] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis: Exploring future development states of software. In *FoSER*, 2010.

[24] R P.L. Buse and W R. Weimer. Automatic documentation inference for exceptions. In *ISSTA*, 2008.

[25] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *ASE*, 2010.

[26] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[27] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP*, 2011.

[28] C.Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. In *Software: Practice and Experience, 34(11)*, pages 1025–1050, 2004.

[29] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *PLDI*, 2013.

[30] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[31] Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *Net.ObjectDays*, 2005.

[32] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: adaptive random testing for object-oriented software. In *ICSE*, 2008.

[33] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2:215–222, May 1976.

[34] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA*, 2007.

[35] Cobertura: a free java tool that calculates the percentage of code accessed by tests. `http://cobertura.sourceforge.net/`.

[36] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[37] Crossword Sage. `http://sourceforge.net/projects/crosswordsage/`.

[38] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, 2006.

[39] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *TOSEM*, 20(4), 9 2011.

[40] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA*, 2010.

[41] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA*, 2006.

[42] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. Automated GUI refactoring and test script repair. In *ETSE*, 2011.

[43] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *ICSE*, 2008.

[44] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *SIGDOC*, 2005.

[45] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, 2006.

[46] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, 2001.

[47] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[48] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *FSE*, 2006.

[49] M D. Ernst, J Cockrell, W G. Griswold, and D Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

[50] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA*, 2003.

[51] David Evans. Static detection of dynamic memory errors. In *PLDI*, 1996.

[52] L Fan, W Xi, and C Yang. API hyperlinking via structural overlap. In *ESEC/FSE*, 2009.

[53] R. Johnson. More details on today's outage. `https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919`.

[54] Volatile and Decentralized. `http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html`.

[55] FireFox. `http://www.mozilla.org/en-US/firefox/new/`.

[56] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI*, 2002.

[57] Freemind. `http://freemind.sourceforge.net`.

[58] Chen Fu, Mark Grechanik, and Qing Xie. Inferring types of references to GUI objects in test scripts. In *ICST*, 2009.

[59] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, 2008.

[60] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *ICSE*, 2008.

[61] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *ICSE*, 2010.

[62] Gantt Project. `http://www.ganttproject.biz/`.

[63] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Using feature locality: can we leverage history to avoid failures during reconfiguration? In *ASAS*, 2011.

[64] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *FSE*, 2004.

[65] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.

[66] Claire Goues and Westley Weimer. Specification mining with few false positives. In *TACAS*, 2009.

[67] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, 2009.

[68] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, 2002.

[69] Alex Groce and Willem Visser. What went wrong: explaining counterexamples. In *SPIN*, 2003.

[70] John C. Grundy. Engineering component-based, user-configurable collaborative editing systems. In *EHCI*, 1999.

[71] Sumit Gulwani and Mark Marron. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.

[72] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *CC*, 2006.

[73] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[74] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. Improved error reporting for software that uses black-box components. In *PLDI*, 2007.

[75] W. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *ISSTA*, 2009.

[76] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[77] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. In *ICSE*, 2009.

[78] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[79] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *SPIN*, 2003.

[80] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *PLDI*, 2009.

[81] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, 1990.

[82] Susan. Horwitz, Thomas. Reps, and David. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.

[83] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[84] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: recording local executions to reproduce concurrency failures. In *PLDI*, 2013.

[85] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing GUI test suites using a genetic algorithm. In *ICST*, 2010.

[86] T-Y Huang, P-C Chou, C-H Tsai, and H-A Chen. Automated fault localization with statistically suspicious program states. In *LCTES*, 2007.

[87] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in Linux and eCos. In *VaMoS*, 2012.

[88] JabRef. `http://jabref.sourceforge.net/`.

[89] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[90] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

[91] Javalanche. `https://github.com/david-schuler/javalanche/`.

[92] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object capture-based automated testing. In *ISSTA*, 2010.

[93] JChord. http://pag.gatech.edu/chord/.

[94] JDK 1.6. http://download.oracle.com/javase/6/docs/api/.

[95] JDK Swing Framework. http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[96] JEdit. http://www.jedit.org/.

[97] D Jeffrey, N Gupta, and R Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[98] Jean-Marc Jézéquel. Reifying variants in configuration management. *ACM TOSEM.*, 8(3):284–295, July 1999.

[99] Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *ICSE*, 2012.

[100] JMeter. http://jmeter.apache.org/.

[101] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, 2005.

[102] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[103] Karen S Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[104] JPF symbolic execution engine. http://javapathfinder.sourceforge.net/.

[105] JSAP project page. http://martiansoftware.com/jsap/.

[106] Parasoft. JTest manuals version 6.0. online manual.

[107] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *FAST*, 2010.

[108] K.Claessen and J.Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[109] Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.

[110] Miryung Kim. *Analyzing and inferring the structure of code change*. PhD thesis, University of Washington, 2008.

[111] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, 2007.

[112] Christos Kloukinas and Valerie Issamy. Automating the composition of middleware configurations. In *ASE*, 2000.

[113] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE*, 2008.

[114] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *SOSP*, 2006.

[115] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[116] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[117] B Lerner, M Flower, D Grossman, and C Chambers. Searching for type-error messages. In *PLDI*, 2007.

[118] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 2012.

[119] Peng Li and Eric Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD*, 2008.

[120] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, 2005.

[121] B Liblit, M Naik, A X. Zheng, A Aiken, and M I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[122] Ben Liblit. *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

[123] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2013.

[124] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *ASE*, 2005.

[125] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[126] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE*, 2005.

[127] LOCC. `http://csdl.ics.hawaii.edu/Plone/research/locc/`.

[128] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *ICSE*, 2012.

[129] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE*, 2008.

[130] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In *FSE*, 2004.

[131] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[132] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, 2003.

[133] Steve McConnell. Code complete, 2nd edition. Microsoft Press, 2004.

[134] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, November 2008.

[135] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *FSE*, 2000.

[136] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *FSE*, 2003.

[137] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, 2012.

[138] Amir Michail and Tao Xie. Helping users avoid bugs in GUI applications. In *ICSE*, 2005.

[139] Microsoft Office Online Help Forum. `http://support.microsoft.com/gp/gp_newsgroups_master`.

[140] Configuration error brings down the Azure cloud platform. `http://www.evolven.com/blog/configuration-error-brings-down-the-azure-cloud\-platform.html`.

[141] MySQL. http://www.mysql.com/.

[142] Adithya Nagarajan and Atif Memon. Refactoring using event-based profiling. In *REFAC*, 2003.

[143] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. Regression testing in the presence of non-code changes. In *ICST*, 2011.

[144] NanoXML. http://nanoxml.sourceforge.net//.

[145] Catherine Oriat. Jartege: a tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, 2005.

[146] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, 2005.

[147] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *ISSTA*, 2008.

[148] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[149] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security*, 2013.

[150] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *ICSE*, 2012.

[151] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[152] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[153] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[154] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *OOPSLA*, 2007.

[155] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, 2008.

[156] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *ASE*, 2010.

[157] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE*, 2009.

[158] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, 2008.

[159] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.

[160] Randoop home. `http://people.csail.mit.edu/cpacheco/randoop/1.2/doc/index.php`.

[161] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *OOPSLA*, 2004.

[162] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[163] Rhino project page. `http://www.mozilla.org/rhino/`.

[164] David Roach, Hal Berghel, and John R. Talburt. An interactive source commenter for prolog programs. In *SIGDOC*, 1990.

[165] Pierre N. Robillard. Schematic pseudocode for program constructs and its computer automation by schemacode. *Commun. ACM*, 29:1072–1089, November 1986.

[166] Jeremias Robler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *ISSTA*, 2012.

[167] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *ASE*, 2005.

[168] David Saff, Marat Boshernitsan, and Michael D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. Technical Report MIT-CSAIL-TR-2008-002, Cambridge, MA, January 14, 2008.

[169] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[170] SAT4J project page. `http://www.sat4j.org/`.

[171] Misconfiguration brings down entire .se domain in Sweden. http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden.

[172] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[173] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, 2007.

[174] The SimMetrics String Similarity Metric Library. http://sourceforge.net/projects/simmetrics/.

[175] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.

[176] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. Relation extraction for inferring access control rules from natural language artifacts. In *ACSAC*, 2014.

[177] Charles Song, Adam Porter, and Jeffrey S. Foster. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *ICSE*, 2012.

[178] Yee Jiun Song, Flavio Junqueira, and Benjamin Reed. Bft for the skeptics. In *Proc. ACM SOSP'09 Work in Progress Session*.

[179] Soot home. http://www.sable.mcgill.ca/soot/.

[180] G Sridhara, E Hill, D Muppaneni, L Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE*, 2010.

[181] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, 2011.

[182] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *PLDI*, 2007.

[183] Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. Finding failure-inducing changes in java programs using change classification. In *FSE*, 2006.

[184] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[185] William N. Sumner, Tao Bao, and Xiangyu Zhang. Selecting peers for execution comparison. In *ISSTA*, 2011.

[186] William N. Sumner and Xiangyu Zhang. Comparative causality: explaining the differences between executions. In *ICSE*, 2013.

[187] Synoptic. `http://code.google.com/p/synoptic/`.

[188] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE*, 2011.

[189] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *FSE*, 2012.

[190] Suresh Thummalapenta, Peli de Halleux, Nikolai Tillmann, and Scott Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *TAP*, 2010.

[191] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *ESEC/FSE*, 2009.

[192] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[193] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.*, 23(4):38–47, 2006.

[194] Time And Money. `http://sourceforge.net/projects/timeandmoney/`.

[195] tinySQL project page. `http://www.jepstone.net/tinySQL/`.

[196] F. Tip. A survey of program slicing techniques. *Journal Of Programming Languages*, 3:121–189, 1995.

[197] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[198] Christopher D. Turner and David J. Robson. The state-based testing of object-oriented programs. In *ICSM*, 1993.

[199] UISpec4J. `http://www.uispec4j.org/`.

[200] G. Venolia, R. DeLine, and T. LaToza. Software development at microsoft observed. *Technical Report MSR-TR-2005-140, Microsoft Research, Redmond, WA, Oct. 2005.*

[201] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.

[202] WALA. http://wala.sourceforge.net.

[203] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[204] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, 2008.

[205] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *FSE*, 2012.

[206] Mark Weiser. Program slicing. In *ICSE*, 1981.

[207] Weka. www.cs.waikato.ac.nz/ml/weka/.

[208] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.

[209] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

[210] Ian. H. Witten, Alistair. Moffat, and Timothy. C. Bell. Managing gigabytes: Compressing and indexing documents and images - morgan kaufmann, 1996.

[211] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, 2013.

[212] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *ASE*, 2013.

[213] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *FSE*, 2012.

[214] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan de Halleux, Michal Moskal, , and Tao Xie. User-aware privacy control via extended static-information-flow analysis. *Automated Software Engineering Journal*, 2014.

[215] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan De Halleux, and Michal Moskal. User-aware privacy control via extended static-information-flow analysis. In *ASE*, 2012.

[216] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS 2005*, April 2005.

[217] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *ASE*, 2003.

[218] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[219] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[220] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Experience mining google's production console logs. In *SLAML*, 2010.

[221] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, June 1991.

[222] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[223] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, September 2004.

[224] Xun Yuan, Myra B. Cohen, and Atif M. Memon. GUI interaction testing: Incorporating event context. *IEEE TSE*, 37(4), 2011.

[225] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE*, 2007.

[226] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, 1999.

[227] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[228] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28:183–200, February 2002.

[229] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *FSE*, 2012.

[230] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*, 2010.

[231] Sai Zhang. Palus: a hybrid automated test generation tool for Java. In *ICSE Companion*, 2011.

[232] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.

[233] Sai Zhang and Michael D. Ernst. Which configuration option should I change? In *ICSE*, 2014.

[234] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, 2008.

[235] Sai Zhang, Hao Lü, and Michael D. Ernst. Automatically repairing broken workflows for evolving GUI applications. In *ISSTA*, 2013.

[236] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, 2011.

[237] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *ASE*, 2013.

[238] Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, 2011.

[239] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.

[240] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.

[241] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.

[242] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *ICML*, 2006.

[243] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with MUT-aware sequence recommendation. In *ASE*, 2010.

[244] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE*, 2004.

# VITA

Sai Zhang holds the B.Eng. and M.Sc. degrees from Shanghai Jiao Tong University (SJTU), and the M.S. degree from the University of Washington. He has previously been a full-time employee at Morgan Stanley. In 2009, he began his Ph.D. studies at University of Washington under the advisement of Michael D. Ernst. His chief research interest is software engineering, in particular software analysis, testing, error diagnosis, evolution, compilation, and programming systems.