Copyright
by
Lingming Zhang
2014

The Dissertation Committee for Lingming Zhang
certifies that this is the approved version of the following dissertation:

Unifying Regression Testing with Mutation Testing

Committee:

Sarfraz Khurshid, Supervisor

Dewayne E. Perry

Darko Marinov

Miryung Kim

Adnan Aziz

Unifying Regression Testing with Mutation Testing

by

Lingming Zhang, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my family.

Acknowledgments

I would like to thank all the people who helped me, accompanied me, and encouraged me during my wonderful stay in Austin.

I want to give my deepest thanks to my Ph.D. advisor Sarfraz Khurshid. I am very fortunate to work under his supervision. He is not only a great researcher, teacher, advisor, but also a true friend. He spent tremendous amount of time and energy to help me with various problems that I faced in my research as well as daily life. When I get stuck in research, he is the person to help me out; even when I locked my key in the car, he is the person to pick me up. At the same time, he also tried his best to train me to be an independent researcher. Simply, I cannot imagine a better advisor.

I want to thank Darko Marinov and Milos Gligoric from University of Illinois at Urbana-Champaign. I really enjoyed working with them, and really appreciate all their kindly help during my Ph.D. study. I will never forget that Darko worked all night to help me revising my research drafts. While officially Darko is not my co-advisor, in my mind, he is always my co-advisor. I will also miss the old days that Milos and I spent weeks together working on some interesting joint projects.

I want to thank Dewayne E. Perry, Miryung Kim, and Adnan Aziz from our software engineering track. Dewayne is a great researcher as well

as a great director. His cheerful personality and focus on research inspire me a lot. Miryung is a great teacher and collaborator. I learnt a lot from her *software evolution* course, and really appreciate all her kind supervision during the course and later in our joint projects. I also thank Adnan for his valuable comments, suggestions, and questions when preparing this dissertation.

I want to thank my previous advisor Lu Zhang and Hong Mei from Peking University, as well as Gregg Rothermel from University of Nebraska at Lincoln and Tao Xie from University of Illinois at Urbana-Champaign. I really appreciate their insightful supervision and helpful comments during the early stage of my graduate study. Without them, I can hardly imagine that I can eventually become a Ph.D. in the software engineering area.

I want to thank all my other collaborators during my Ph.D.'s study – Xiaoyin Wang (University of Texas at San Antonio), Dan Hao (Peking University), Guowei Yang (Texas State University), Neha Rungta (NASA Ames), Suzette Person (NASA Langley), Cristiano Pereira and Gilles Pokam (Intel Labs). I really enjoyed working with them, and have learnt a lot from them.

I am proud to be a member of the Software Verification Validation and Testing group. It is my great pleasure to work with the very brilliant fellow students – Shadi Abdul Khalek, Junaid Haroon Siddiqui, Muhammad Zubair Malik, Razieh Nokhbeh Zaeem, Divya Gopinath, Chang Hwan Peter Kim, Guowei Yang, Sam Harwell, Diego Funes, Shounak Roychowdhury, Rui Qiu, Shiyu Dong, Allison Sullivan, and Raghavendra Srinivasan. In addition, I will miss my friends in the software engineering track – Yuqun Zhang, Ripon Saha,

Yen-Jung Chang, James (Xi) Zheng, Meiru Che, Lisa (Jinru) Hua, Wei-Lun Hung, and Tianyi Zhang.

I thank the helpful staff at the Electrical and Computer Engineering Department, especially Melanie Gulick and RoseAnna Goewey.

This work would not have been possible without the support from my family and my friends. I especially thank my parents, my younger brother, and my girl friend for their love, encouragement, and trust. I dedicate this dissertation to them.

The work presented in this dissertation is partially supported by NSF awards under Grant Nos. CCF-0845628, CNS-0958231, CNS-0958199, CCF-0746856, and IIS-0438967, AFOSR grant FA9550-09-1-0351, Chiang Chen Oversea PHD Fellowship, and Google Summer of Code 2013. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of these organizations.

Unifying Regression Testing with Mutation Testing

Publication No. ______________

Lingming Zhang, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid

Software testing is the most commonly used methodology for validating quality of software systems. Conceptually, testing is simple, but in practice, given the huge (practically infinite) space of inputs to test against, it requires solving a number of challenging problems, including evaluating and reusing tests efficiently and effectively as software evolves. While software testing research has seen much progress in recent years, many crucial bugs still evade state-of-the-art approaches and cause significant monetary losses and sometimes are responsible for loss of life.

My thesis is that a unified, bi-dimensional, change-driven methodology can form the basis of novel techniques and tools that can make testing significantly more effective and efficient, and allow us to find more bugs at a reduced cost. We propose a novel unification of the following two dimensions of change: (1) real manual changes made by programmers, e.g., as commonly used to support more effective and efficient *regression testing* techniques; and

(2) mechanically introduced changes to code or specifications, e.g., as originally conceived in *mutation testing* for evaluating quality of test suites. We believe such unification can lay the foundation of a scalable and highly effective methodology for testing and maintaining real software systems.

The primary contribution of my thesis is two-fold. One, it introduces new techniques to address central problems in both regression testing (e.g., *test prioritization*) and mutation testing (e.g., *selective mutation testing*). Two, it introduces a new methodology that uses the foundations of regression testing to speed up mutation testing, and also uses the foundations of mutation testing to help with the fault localization problem raised in regression testing. The central ideas are embodied in a suite of prototype tools. Rigorous experimental evaluation is used to validate the efficacy of the proposed techniques using a variety of real-world Java programs.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xv**

List of Figures **xvii**

Chapter 1. Introduction **1**

1.1 Problem Context 1
1.2 This Thesis 2
1.2.1 Regression Test Prioritization (ICSE'13) 4
1.2.2 Selective Mutation Testing (ASE'13) 5
1.2.3 Test Selection for Mutation Testing (ISSTA'12) 6
1.2.4 Test Prioritization and Reduction for Mutation Testing (ISSTA'13) 6
1.2.5 Mutation Testing for Regression Fault Localization (OOPSLA'13) 7
1.3 Contributions 8
1.4 Organization 10

Chapter 2. Regression Test Prioritization **12**

2.1 Background 12
2.2 Example 14
2.3 Approach 15
2.3.1 Basic Model 16
2.3.2 Extended Model 19
2.3.3 Differentiating p Values 21
2.4 Implementation 23

2.5 Experimental Study 23
2.5.1 Independent Variables 24
2.5.2 Dependent Variable 25
2.5.3 Subject Programs, Test Suites, and Faults 26
2.5.4 Experiment Procedure 27
2.5.5 Threats to Validity 28
2.5.6 Results and Analysis 29
2.5.6.1 RQ1: Existence of Better Strategies Between the Total and Additional Strategies 31
2.5.6.2 RQ2: Impact of Coverage and Test-Case Granularities 34
2.5.6.3 RQ3: Using Differentiated p Values 38
2.5.7 Implications 40
2.6 Summary 41

Chapter 3. Selective Mutation Testing 43
3.1 Background 43
3.2 Study Approach 45
3.2.1 Problem Definition 45
3.2.2 Measurement 46
3.2.3 Combining Operator-Based and Random Mutant Selection 49
3.3 Empirical Study 51
3.3.1 Subject Programs 51
3.3.2 Experimental Design 53
3.3.2.1 Independent Variables 53
3.3.2.2 Dependent Variables 53
3.3.2.3 Experimental Setup 54
3.3.3 Results and Analysis 57
3.3.3.1 Effectiveness for Adequate Test Suites 57
3.3.3.2 Predictive Power for Non-Adequate Test Suites 60
3.3.3.3 Savings Obtained by Mutation Sampling 66
3.3.4 Below 5% 68
3.3.5 Threats to Validity 69
3.4 Summary 70

Chapter 4. Test Selection for Mutation Testing **73**
4.1 Background . . . 73
4.2 Definitions . . . 75
4.2.1 Mutation Testing . . . 75
4.2.2 Regression Testing . . . 77
4.2.3 Regression Mutation Testing . . . 79
4.3 Example . . . 81
4.4 Approach . . . 86
4.4.1 Overview . . . 86
4.4.2 Preprocessing . . . 87
4.4.2.1 Mutant Mapping . . . 88
4.4.2.2 Dangerous-Edge Reachability Analysis . . . 88
4.4.3 ReMT Algorithm . . . 91
4.4.3.1 Basic Algorithm . . . 91
4.4.3.2 Dangerous-Edge Reachability Checking . . . 93
4.4.4 Mutation-Specific Test Prioritization . . . 94
4.4.5 Discussion and Correctness . . . 95
4.5 Implementation . . . 98
4.6 Experimental Study . . . 99
4.6.1 Research Questions . . . 100
4.6.2 Independent Variables . . . 100
4.6.3 Dependent Variables . . . 101
4.6.4 Subjects and Experimental Setup . . . 101
4.6.5 Results and Analysis . . . 102
4.6.5.1 RQ1: Full Mutation Testing Scenario . . . 102
4.6.5.2 RQ2: Partial Mutation Testing Scenario . . . 104
4.6.5.3 RQ3: Mutation-Specific Test Prioritization . . . 107
4.7 Summary . . . 108

Chapter 5. Test Prioritization and Reduction for Mutation Testing 110

5.1 Background 110
5.2 Example 113
5.3 Approach 116
5.3.1 Coverage-Based Initial Test Ordering 117
5.3.2 Power-Based Adaptive Test Ordering 118
5.3.3 Test prioritization 121
5.3.4 Test reduction 123
5.4 Experimental Study 124
5.4.1 Research Questions 124
5.4.2 Independent Variables 125
5.4.3 Dependent Variables 126
5.4.4 Subjects and Experimental Setup 127
5.4.5 Result Analysis 128
5.4.5.1 RQ1: FaMT Test Prioritization 128
5.4.5.2 RQ2: FaMT Test Reduction 137
5.4.5.3 RQ3: Comparison with Regression Techniques . 141
5.4.5.4 RQ4: FaMT Efficiency 143
5.4.5.5 Threats to Validity 144
5.5 Summary 145

Chapter 6. Mutation Testing for Fault Localization in Regression Testing 146

6.1 Introduction 146
6.2 Example 153
6.3 Approach 157
6.3.1 Change Mapping Inference 159
6.3.1.1 Inference for Changed/Deleted Elements 160
6.3.1.2 Inference for Added Elements Overridding/Hiding Existing Elements 161
6.3.1.3 Inference for Added Elements Sharing Overriding/Hiding Hierarchy with Deleted Elements . . 163
6.3.1.4 Inference for Other Added Elements 166

6.3.2 Mutant Suspiciousness Calculation 168
6.3.3 Suspiciousness Combination 171
6.3.4 Tackling the Cost of FIFL: Edit-Oriented Mutation Testing 174
6.4 Implementation 175
6.5 Experimental Study 176
6.5.1 Independent Variables 176
6.5.2 Dependent Variables 177
6.5.3 Subjects and Experimental Setup 178
6.5.4 Results and Analysis 181
6.5.4.1 Overall comparison between FaultTracer and various strategies of FIFL 181
6.5.4.2 Detailed comparison between FaultTracer and FIFL with the default settings 187
6.5.4.3 Discussions 193
6.5.5 Threats to Validity 197
6.6 Summary 198

Chapter 7. Related Work **200**
7.1 Regression Testing 200
7.1.1 Test Selection 200
7.1.2 Test Prioritization 201
7.1.3 Test Reduction 202
7.2 Mutation Testing 204
7.2.1 Reducing Cost of Mutation Testing 204
7.2.2 Detecting Equivalent Mutants 206
7.2.3 Applications of Mutation Testing 207

Chapter 8. Conclusion **209**

Bibliography **211**

List of Tables

2.1 Statistics on Objects of Study 27
2.2 Fisher's LSD test for comparing strategies in the basic model to the *additional* strategy 30
2.3 Fisher's LSD test for comparing strategies in the extended model to the *additional* strategy 30
2.4 Fisher's LSD test for comparing p-differentiated techniques with p-uniform techniques at the test-method level 38
2.5 Fisher's LSD test for comparing p-differentiated techniques with p-uniform techniques at the test-class level 38

3.1 Subject programs used in the evaluation 51
3.2 Selected mutation scores (%) achieved by the test suites that achieve 100% sampled mutation scores 58
3.3 R^2 and τ correlation values between mutation scores on sampled 5% mutants and on mutants before sampling 63
3.4 Selective and sampling mutation testing time 67
3.5 Results of sampling below 5% of selected mutants 69

4.1 Mutants for illustration. 83
4.2 Incrementally collecting full matrix. 83
4.3 Incrementally collecting partial matrix. 83
4.4 Subjects overview. 102
4.5 Experimental results of Javalanche and ReMT under the full mutation testing scenario. 105

5.1 Traditional mutation testing 114
5.2 FaMT test prioritization 114
5.3 FaMT test reduction 114
5.4 Subjects 124
5.5 Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of all neighbor mutants (P_1) 130

5.6 Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of only killed neighbor mutants (P_2) 130

5.7 Execution reduction (%) for killed/all mutants by theoretical techniques and an FaMT prioritization technique. 133

5.8 Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of all neighbor mutants (P_1) 136

5.9 Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of killed neighbor mutants (P_2) . . 136

5.10 Reduction error rates (%) for example FaMT reduction techniques and corresponding random techniques 136

5.11 Comparison between FaMT and regression test prioritization and reduction 141

5.12 Runtime overhead by FaMT techniques 143

6.1 Suspiciousness Calculation for Developer Edits and Mutation Changes. 155

6.2 Subjects overview. 176

6.3 Version pairs with test failures. 179

6.4 Wilcoxon tests for comparing FIFL techniques with FaultTracer 187

6.5 Comparison between FaultTracer and default settings of FIFL. 190

6.6 Summary results when using the default *R50* strategy to rank all edits and rank edits for each failed test 195

7.1 Regression testing areas and their applications for mutation testing 200

List of Figures

1.1 My main PhD research work 3

2.1 Results for test suites at the test-method level with method coverage 29
2.2 Results for test suites at the test-method level with statement coverage 29
2.3 Results for test suites at the test-class level with method coverage 29
2.4 Results for test suites at the test-class level with statement coverage 31
2.5 Prioritization results for models embodied with differentiated p values for each method 34

3.1 Sampling mutation score vs. Selected mutation score, with best fit line (black color) and smoothing spline line (red color), for CommonsLang subject, Meth strategy, and three different sampling ratios 61
3.2 Correlation values for CommonsLang subject, all strategies, and all rates 64

4.1 Example code evolution and test suite. 82
4.2 Inter-procedural CFG for the example. 84
4.3 General approach of ReMT. 86
4.4 Reduction of executions achieved by ReMT and MTP under the partial mutation testing scenario. 106

5.1 Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_1 formula 132
5.2 Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_2 formula 132
5.3 Reduction and error rate trends on different `Threshold` and `MinRatio` values by FaMT reduction 135

6.1 (a) Example in evolution. Note that methods/fields in box are added, methods/fields with line-through are deleted, and methods/fields with underlines are changed. The statements with underlines inside changed methods are added. (b) Tests for the example. . . . 152

6.2 FIFL architecture. . . . 158

6.3 Rules for inferring change mapping. . . . 159

6.4 Code snippet of *Com-Lang* V3.03 to illustrate change mapping for `CM` edits . . . 160

6.5 Code snippet of *Joda-Time* V1.20 to illustrate change mapping for `AM` edits with overridden methods . . . 163

6.6 Illustration for mutant mapping. . . . 164

6.7 Code snippets of *Joda-Time* V1.10 and V1.20 to illustrate change mapping of `AM` edits with deleted methods sharing the same overriding hierarchy . . . 165

6.8 Code snippet of *XStream* V1.21 to illustrate change mapping for `AM` edits without methods sharing the same method overriding hierarchy . . . 167

6.9 Ranking failure-inducing edits using various techniques with various formulae. . . . 181

Chapter 1

Introduction

Software testing continues to be the dominant methodology for validating quality of software. Despite its increasingly important role in reducing the cost of software failures, testing itself remains expensive – not just in terms of the human effort but also in terms of the computational resources. For example, an industrial collaborator of Rothermel et al. reported that running the full *regression* test suites for one of their products cost seven weeks [117].

1.1 Problem Context

The focus of our work is on two specific areas in software testing: *regression testing* [39, 47, 50, 51, 67, 88, 102, 114, 117, 148, 150, 154, 159] – where the key problem is how to effectively and efficiently test a *new* version of a progam that evolved (i.e., underwent some changes or edits that are made by the developer) – and *mutation testing* [6, 32, 42, 46, 83, 122, 149, 156] – where the key problem is how to accurately determine the quality of a test suite (i.e., in terms of its ability to find bugs).

Regression testing contains three main research areas: (1) *test prioritization*, (2) *test reduction*, and (3) *test selection*. Test prioritization [39,

88, 117, 148, 159] reorders tests to reveal regression faults faster. Test reduction [47, 50, 67] aims to reduce *redundant* tests to make regression testing more efficient. Test selection [51, 102, 114, 150] only executes the subset of tests that are influenced by program edits.

Mutation testing has two basic steps. One, generate desired variants (known as *mutants*) of the original program under test through small syntactic transformations (known as *mutation operators*). Two, execute the generated mutants against a test suite to check whether the test suite can distinguish the behavior of the mutants from the original program (known as *killing* the mutants). The more mutants the test suite can kill, the more effective the test suite is considered to be. Mutation testing has been viewed as the strongest test criterion in terms of characterizing high-quality test suites [11, 41]. Researchers have used mutation testing in numerous studies on software testing; see a recent survey by Jia and Harman [60]. Some studies have even shown that mutation testing can be more suitable than manual fault seeding in simulating real program faults for software testing experimentation [13, 14, 36].

1.2 This Thesis

My thesis is that a *unified*, *bi-dimensional*, *change-driven* methodology can form the basis of novel techniques and tools that can make testing significantly more effective and efficient, and allow us to find more bugs at a reduced cost. We propose a novel unification of the following two dimensions of change: (1) *real code changes* made by programmers, e.g., as commonly

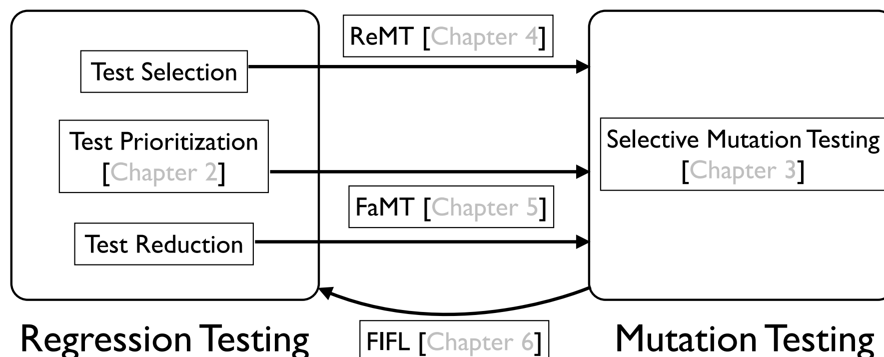


Figure 1.1: My main PhD research work

analyzed to support more effective and efficient regression testing techniques, and (2) *mechanically introduced code changes*, e.g, as originally conceived in mutation testing for evaluating quality of test suites; and We believe such unification can lay the foundation of a scalable and highly effective methodology for testing and maintaining real software systems.

The primary contribution of my thesis research is two-fold. One, it introduces new techniques to address central problems in regression testing (e.g., *test prioritization*) and mutation testing (e.g., *selective mutation testing*). Two, it introduces a new methodology that uses the foundations of regression testing to speed up mutation testing, and also uses the foundations of mutation testing to help with the fault localization in regression testing.

Figure 1.1 gives an overview of this dissertation and how the five main chapters, which present the key technical ideas, relate to the areas of regression testing and mutation testing. Chapter 2 presents a technique for test prioritization within the traditional area of regression testing [148]. Chap-

ter 3 presents a technique for selective mutation testing within the traditional area of mutation testing [147]. Chapter 4 presents a technique inspired by regression test selection for mutation testing [155]. Chapter 5 presents a technique inspired by test prioritization and reduction for mutation testing [153]. Chapter 6 presents an application of mutation testing for more precise fault localization in regression testing [158]. The subsections that follow give a brief overview of these five chapters.

1.2.1 Regression Test Prioritization (ICSE'13)

In recent years, researchers have intensively investigated various topics in test-case prioritization, which aims to reorder test cases to increase the rate of fault detection during regression testing. The *total* and the *additional* prioritization strategies, which prioritize based on total numbers of elements covered per test, and numbers of additional (not-yet-covered) elements, are two widely-adopted generic strategies used for such prioritization. Chapter 2 presents a basic model and an extended model [148] that unify the total strategy and the additional strategy. Both models yield a spectrum of generic strategies ranging between the *total* and *additional* strategies, depending on a parameter referred to as the p value. Chapter 2 also introduces four heuristics to obtain differentiated p values for different methods under test. The empirical study on 19 versions of four real-world Java programs demonstrates that a wide ranges of strategies in both the basic and extended models with uniform p values can significantly outperform both the *total* and *additional* strategies.

In addition, the empirical results also demonstrate that using differentiated p values for both the basic and extended models with method coverage can even outperform the *additional* strategy using statement coverage.

1.2.2 Selective Mutation Testing (ASE'13)

Mutation testing is a powerful methodology for evaluating the quality of a test suite. However, the methodology is also very costly, as the test suite may have to be executed for each mutant. *Selective mutation testing* is a well-studied technique to reduce this cost by selecting a subset of all mutants, which would otherwise have to be considered in their entirety. Two common approaches are *operator-based mutant selection*, which only generates mutants using a subset of mutation operators, and *random mutant selection*, which selects a subset of mutants generated using all mutation operators. While each of the two approaches provides some reduction in the number of mutants to execute, applying either of the two to medium-sized, real-world programs can still generate a huge number of mutants, which makes their execution too expensive. Chapter 3 presents eight random sampling strategies defined on top of operator-based mutant selection, and empirically validates that operator-based selection and random selection can be applied in tandem to further reduce the cost of mutation testing [147][1]. The experimental results show that even sampling only 5% of mutants generated by operator-based selection

[1]Note that this is a joint work [147] with another PhD student, Milos Gligoric, from University of Illinois at Urbana-Champaign.

can still provide precise mutation testing results, while reducing the average mutation testing time to 6.54% (i.e., on average less than 5 minutes for this study).

1.2.3 Test Selection for Mutation Testing (ISSTA'12)

Chapter 4 presents *Regression Mutation Testing (ReMT)* [155], a new technique to speed up mutation testing for evolving systems. The key novelty of ReMT is to incrementally calculate mutation testing results for the new program version based on the results from the old program version; ReMT uses a static analysis to check which results can be safely reused. ReMT also employs a mutation-specific test prioritization to further speed up mutation testing. The chapter also presents an empirical study on six evolving systems, whose sizes range from 3.9KLoC to 88.8KLoC. The empirical results show that ReMT can substantially reduce mutation testing costs, indicating a promising future for applying mutation testing on evolving software systems.

1.2.4 Test Prioritization and Reduction for Mutation Testing (ISSTA'13)

The central idea behind the mutation testing approach is to generate *mutants*, which are small syntactic transformations of the program under test, and then to measure for a given test suite how many mutants it *kills*. A test t is said to kill a mutant m of program p if the output of t on m is different from the output of t on p. The effectiveness of mutation testing in determining the quality of a test suite relies on the ability to apply it

using a large number of mutants. However, running many tests against many mutants is time consuming. Chapter 5 presents a family of techniques to reduce the cost of mutation testing by prioritizing and reducing tests to more quickly determine the sets of killed and non-killed mutants [153]. The chapter also includes an extensive experimental study to show the effectiveness and efficiency of the proposed techniques.

1.2.5 Mutation Testing for Regression Fault Localization (OOPSLA'13)

Chapter 6 presents a novel methodology for localizing regression faults in code as it evolves [158]. The insight is that the essence of failure-inducing edits made by the developer can be captured using mechanical program transformations (e.g., mutation changes). Based on the insight, the chapter presents the FIFL framework, which uses both the spectrum information of edits (obtained using the existing FaultTracer approach [150–152]) as well as the potential impacts of edits (simulated by mutation changes) to achieve more accurate fault localization. The effectiveness of FIFL was evaluated on real-world repositories of nine Java projects ranging from 5.7KLoC to 88.8KLoC. The experimental results show that FIFL is able to outperform the state-of-the-art FaultTracer technique for localizing failure-inducing program edits significantly. For example, all 19 FIFL strategies that use both the spectrum information and simulated impact information for each edit outperform the existing FaultTracer approach statistically at the significance level of 0.01. In addition, FIFL with its default settings outperforms FaultTracer by 2.33% to

86.26% on 16 of the 26 studied version pairs, and is only inferior than FaultTracer on one version pair.

1.3 Contributions

This dissertation makes the following contributions [147, 148, 153, 155, 158]:

- **Unifying mutation testing and regression testing.** This dissertation introduces the idea of unifying regression testing with mutation testing – two well researched methodologies that previous work has explored independently – to make each of the methodologies more effective or efficient (Figure 1.1). We believe such a unified view will spur more research work in each of the two areas as well as their further unification.
- **Unified models for test prioritization.** This dissertation proposes a new approach that creates better prioritization techniques by controlling the uncertainty of fault detection capability in test-case prioritization, and presents two models that unify the *total* and *additional* strategies and can also yield a spectrum of more effective strategies having flavors of both the *total* and *additional* strategies.
- **Advanced selective mutation testing.** This dissertation proposes *sampling mutation testing*, which further reduces mutation testing cost by applying operator-based mutant selection and random mutant selection in tandem, and presents eight sampling strategies that can further

reduce the mutation testing cost by 20 times without losing much accuracy.

- **Test selection for mutation testing.** This dissertation introduces of idea of using test selection to make mutation testing of evolving systems more efficient, and presents a core technique for regression mutation testing (ReMT) using dangerous-edge reachability analysis inspired by regression test selection.

- **Test prioritization for mutation testing.** This dissertation introduces the general idea of optimizing mutation testing using test prioritization, and presents a family of test prioritization techniques for mutation testing based on coverage information as well as history of test executions on other mutants (e.g., the accumulating number of mutants that the test killed or did not kill before executing the current mutant).

- **Test reduction for mutation testing.** The cost of mutation testing has two key elements – executing some tests for killed mutants and executing every test for non-killed mutants. As test prioritization only addresses the first element, this dissertation also presents a family of test reduction techniques for mutation testing, which can effectively reduce the number of test executions for all mutants without losing much accuracy.

- **Mutation testing for regression fault localization.** This dissertation introduces the mutation testing methodology to the realm of localiz-

ing failure-inducing program edits in regression testing. This dissertation combines two dimensions of changes to calculate the coverage spectra as well as impacts of program edits to improve fault localization for evolving software. In other words, this dissertation initializes the idea of localizing faulty edits based on fault injection.

- **Experimental studies.** We evaluated our completed work on various real-world Java programs from open-source as well as the well-known Software-artifact Infrastructure Repository (SIR) [35]. The experimental results demonstrate the effectiveness and efficiency of the proposed work.

1.4 Organization

The rest of this document is organized as follows.

Chapter 2 describes our work on regression test prioritization, which unifies the traditional *total* and *additional* strategies and provides a spectrum of (more effective) techniques between the *total* and *additional* strategies for test prioritization.

Chapter 3 describes our work on selective mutation testing, which applies operator-based and random mutant selection in tandem to further speed up mutation testing.

Chapter 4 presents our regression mutation testing approach (ReMT), which incrementally collects mutation testing results based on the differences between two program versions.

Chapter 5 presents our FaMT approach, which further applies test prioritization and reduction techniques to the area of mutation testing.

Chapter 6 presents our FIFL approach, which uses mechanical changes to simulate the impact of real programer changes, and utilizes the simulated impact information to help with the diagnois of failure-inducing changes.

Chapter 7 discusses the related work in both the regression testing and mutation testing areas.

Finally, Chapter 8 concludes this disseration.

Chapter 2

Regression Test Prioritization

Before stepping into the unification of regression testing and mutation testing, this dissertation first presents my research work in each of the traditional regression testing and mutation testing areas. This chapter presents my approach for test prioritization in the traditional regression testing area, which was presented at the 35th IEEE/ACM International Conference on Software Engineering (ICSE 2013) [148].

2.1 Background

Software engineers usually maintain a large number of test cases, which can be reused in regression testing to test software changes. Due to the large number of test cases, regression testing can be very time consuming. Test-case prioritization [37–39, 108, 117, 135], which attempts to re-order regression test cases to detect faults as early as possible, has been intensively investigated as a way to deal with lengthy regression testing cycles.

In test-case prioritization, a fundamental topic involves prioritization strategies. In previous work, researchers have studied two greedy strategies (the *total* and *additional* strategies), which are generic for different coverage

criteria. Given a coverage criterion, the *total* strategy sorts test cases in descending order of coverage, whereas the *additional* strategy always picks a next test case having the maximal coverage of items not yet covered by previously prioritized test cases. In addition to these two strategies, researchers have also investigated other generic strategies. Li et al. [76] investigated the 2-optimal greedy strategy [78], a hill-climbing strategy, and a genetic programming strategy. Jiang et al. [61] investigated adaptive random prioritization. Their empirical results show that the *additional* strategy remains the most effective generic strategy on average in terms of rate of fault detection.

There is also, however, a weakness in the *additional* strategy. Consider statement coverage for instance. In the *additional* strategy, after a test case t is chosen, no statement covered by t is explicitly considered again until all coverable statements are covered at least one time. As a result, when there is a fault f in one statement covered by t but not covered by any test case chosen before t, if t cannot detect f, the detection of f may be greatly postponed. In contrast, the *total* strategy does not have this weakness, because when choosing a next test case, the *total* strategy always considers all statements no matter whether or not previously chosen test cases have covered the statements. This said, as the *total* strategy counts only the numbers of statements covered by each test case, it may be more inclined to choose test cases to cover statements previously covered many times than to choose test cases to cover previously not (intensively) covered statements. Thus, the *total* strategy may postpone the detection of faults in rarely covered statements. As a result, **our insight**

is that some strategy that has the flavor of both the *additional* strategy and the *total* strategy may be more advantageous.

In this chapter, we propose a unified view (including a basic model and an extended model) for generic strategies in test-case prioritization. In our models, the *total* and *additional* strategies are extreme instances, and the models also define various generic strategies that lie between the *total* strategy and the *additional* strategy depending on the value of fault detection probability (referred to as the p value). In addition, we further extend the models by using differentiated p values. We view our models as an initial framework to control the uncertainty of fault detection during test-case prioritization, and believe more techniques can be derived based on our models. We performed an empirical study to compare our strategies with the *total* strategy and the *additional* strategy. Our results demonstrate that many of our strategies can outperform both the *total* and *additional* strategies.

2.2 Example

Shown in Section 2.1, in the state-of-the-art *additional* strategy, after a test case t is chosen, no statement covered by t is explicitly considered again until all coverable statements are covered at least one time. As a result, when there is a fault f in one statement covered by t but not covered by any test case chosen before t, if t cannot detect f, the detection of f may be greatly postponed. To understand the situation in which a test case covers a statement but does not reveal a fault in the statement, consider the following piece of

code with a fault in line 5. The code is a method returning the larger of x and y. A test case in which the value of x is smaller than that of y covers the faulty statement and detects the fault. However, a test case in which the value of x is equal to that of y also covers the faulty statement but does not detect the fault.

```
1   int max(int x, int y) {
2       if (x>y)
3           return x;
4       else
5           return x;//should be "return y".
6   }
```

In fact, research on test-suite reduction [57, 58, 115, 143] has demonstrated that re-covering already covered statements may enhance fault-detection capability. Furthermore, when we consider test-case prioritization based on coverage information at a coarser level (e.g., the method level), it may be more common for a test case to miss a fault in a method covered by the test case, because that test case may fail to cover the faulty statement in the method. This motivates our unified models of explicitly considering the probability of fault detection in test prioritization.

2.3 Approach

With the *additional* strategy, the primary concern is to cover units not yet covered by previous test cases. This strategy should be well suited for circumstances in which the probability of a test case detecting faults in units it covers is high. On the other hand, the primary concern for the *total* strategy is to cover the most units with each test case. This strategy should be well suited

for circumstances in which the probability of a test case detecting faults in units it covers is low. Thus, if we explicitly consider the probability for a test case to detect faults in units it covers, we may devise strategies to take advantage of the strengths of both the *total* and *additional* strategies. More specifically, our models initially assign probability values for each program *unit*[1]. Then, each time a *unit* is covered by a test case (that could potentially detect some fault(s) in the *unit*), the probability that the *unit* contains undetected faults is reduced by some ratio between 0% (as in the *total* strategy) and 100% (as in the *additional* strategy). In this way, we build a spectrum of generic prioritization strategies between the *total* and *additional* strategies.

2.3.1 Basic Model

In our basic model, when a test case t covers a unit u, we refer to the probability that t can detect faults in u as p. Consider a test suite $T = \{t_1, t_2, ..., t_n\}$ containing n test cases and a program $U = \{u_1, u_2, ..., u_m\}$ containing m units. Algorithm 1 depicts test-case prioritization in our basic model, in which we use a Boolean array $Cover[i, j]$ $(1 \leq i \leq n, 1 \leq j \leq m)$ to denote whether test case t_i covers unit u_j.

In Algorithm 1, we use an array $Prob[j]$ $(1 \leq j \leq m)$ to store the probability that unit u_j contains undetected faults. Initially, we set the value

[1]As our approach is intended to work with different coverage criteria, we use *unit* as a generic term to denote different structural elements used in different coverage criteria. For example, a *unit* represents a *statement* for statement coverage but a *method* for method coverage.

Algorithm 1: Prioritization in the basic model with p

```
1: for each j (1 ≤ j ≤ m) do
2:     Prob[j] ← 1
3: end for
4: for each i (1 ≤ i ≤ n) do
5:     Selected[i] ← false
6: end for
7: for each i (1 ≤ i ≤ n) do
8:     k ← 1
9:     while Selected[k] do
10:        k ← k + 1
11:    end while
12:    sum ← 0
13:    for each j (1 ≤ j ≤ m) do
14:        if Cover[k, j] then
15:            sum ← sum + Prob[j]
16:        end if
17:    end for
18:    for each l (k + 1 ≤ l ≤ n) do
19:        if not Selected[l] then
20:            s ← 0
21:            for each j (1 ≤ j ≤ m) do
22:                if Cover[l, j] then
23:                    s ← s + Prob[j]
24:                end if
25:            end for
26:            if s > sum then
27:                sum ← s
28:                k ← l
29:            end if
30:        end if
31:    end for
32:    Priority[i] ← k
33:    Selected[k] ← true
34:    for each j (1 ≤ j ≤ m) do
35:        if Cover[k, j] then
36:            Prob[j] ← Prob[j] * (1 − p)
37:        end if
38:    end for
39: end for
```

of $Prob[j]$ $(1 \leq j \leq m)$ to be 1. We use a Boolean array $Selected[i]$ $(1 \leq i \leq n)$ to store whether test case t_i has been selected for prioritization. Initially, we set the value of $Selected[i]$ $(1 \leq i \leq n)$ to be $false$. Furthermore, we use an array $Priority[i]$ $(1 \leq i \leq n)$ to store the prioritized test cases. If $Priority[i]$ is equal to k $(1 \leq i, k \leq n)$, test case t_k is ordered in the ith position.

In Algorithm 1, lines 1-6 perform initialization. In the main loop from lines 7 to 39, each iteration determines which test case to place in the prioritized test suite. Lines 8-31 find a test case t_k such that t_k is previously not chosen and the sum of probabilities that units covered by t_k contain undetected faults is the highest among test cases not yet chosen. Note that, as the basic model utilizes a *uniform* probability p for fault detection in covered units, lines 8-31 actually find a test case with the highest probability of detecting previously undetected faults. In particular, lines 8-17 find the first test case t_k not previously chosen for prioritization and calculate the sum of the probabilities that units covered by t_k contain undetected faults, and lines 18-31 check whether there is another unchosen test case t_l for which the sum of the probabilities that the covered units contain undetected faults is higher than that for t_k. Line 32 sets the ith position in the prioritized test suite to t_k, and line 33 marks t_k as already chosen for prioritization. Lines 34-38 update the probability that units contain undetected faults for each unit covered by t_k.

Algorithm 1 is in fact a variant of the algorithm for the *additional* strategy. The main difference is that this algorithm tries to find the test case covering units with the maximal sum of probabilities of containing undetected faults. Due to the similarity between this algorithm and the *additional* strategy, its worst case time cost is the same as that of the *additional* strategy (i.e., $O(mn^2)$, where n is the number of test cases and m is the number of units [117]).

With Algorithm 1, an optimistic tester who believes that a test case is likely to detect faults in covered units may set the value of p to 1. In such a circumstance, this algorithm is equivalent to the *additional* strategy. The reason for this is that lines 34-38 set the probability for any previously covered unit to contain undetected faults to 0. In contrast, a pessimistic tester who is concerned with the situation in which a test case may not detect faults in units covered by the test case may set the value of p to 0. In such a circumstance, this algorithm is equivalent to the *total* strategy. The reason is that lines 34-38 do not change the probability that any previously covered unit contains undetected faults. Note that setting p to 0 does not render the algorithm as efficient as the original *total* strategy, whose worst case time cost is $O(mn)$ [117]. Finally, if a tester sets the value of p to a number between 0 and 1, this algorithm is a strategy between the *total* strategy and the *additional* strategy. The closer p is to 0, the closer this algorithm is to the *total* strategy; and the closer p is to 1, the closer this algorithm is to the *additional* strategy.

2.3.2 Extended Model

In our basic model and previously proposed strategies for test-case prioritization, when a test case t covers a unit u, the number of times t covers u is not further considered. That is, no matter how many times t covers u, the algorithm treats u as having been covered once. Intuitively, the more times t covers u, the more probable it may be for t to detect faults in u. Therefore, considering the ability of a test case to cover a unit multiple times may result

in higher effectiveness.

We now extend our basic model to consider multiple coverage of units by given test cases. In our extended model, the main body of the algorithm is the same as the algorithm in our basic model, but the extended algorithm uses a different method for calculating the probability for a test case to detect previously undetected faults. We extend $Cover[i, j]$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$ to denote the number of times test case t_i covers unit u_j. We now present the main differences between the two algorithms.

First, as the number of times test case t_k covers unit u_j is $Cover[k, j]$, the probability for unit u_j to contain undetected faults changes from $Prob[j]$ to $Prob[j] * (1-p)^{Cover[k,j]}$ after executing t_k if we consider each instance of coverage to have an equal probability p of detecting faults. That is to say, for unit u_j alone, execution of t_k decreases the probability that u_j contains undetected faults by $Prob[j] * (1-(1-p)^{Cover[k,j]})$. Thus, in the extended algorithm, we change lines 15 and 23 of Algorithm 1 to $sum \leftarrow sum + Prob[j] * (1-(1-p)^{Cover[k,j]})$ and $s \leftarrow s + Prob[j] * (1-(1-p)^{Cover[l,j]})$, respectively.

Second, after we select test case t_k for prioritization at the ith place, the probability for unit u_j to contain undetected faults changes from $Prob[j]$ to $Prob[j] * (1-p)^{Cover[k,j]}$. Thus, in the extended algorithm, we change line 36 of Algorithm 1 to $Prob[j] \leftarrow Prob[j] * (1-p)^{Cover[k,j]}$. The worst case time cost of the extended algorithm is also $O(mn^2)$, the same as that of Algorithm 1.

In the extended algorithm, if we set p to 1, the algorithm is the same

as the *additional* strategy, because $(1-p)^{Cover[k,j]}$ is equal to 0 when p is equal to 1. However, if we set p to 0, the extended algorithm cannot distinguish any test cases from each other,[2] because $1-(1-p)^{Cover[k,j]}$ is always equal to 0 when p is equal to 0. If we set p to a number between 0 and 1, the extended algorithm also represents a strategy between the *total* and *additional* strategies, considering multiple coverage for each test case.

2.3.3 Differentiating p Values

In Section 2.3.1, in our basic model, whenever a test case t covers a unit u, we consider the probability for t to detect faults in u to be uniformly p. In Section 2.3.2, using our extended model, we reason that when t covers u multiple times, the probability for t to detect faults in u may not be uniform, but each instance of coverage also implies a uniform probability of fault detection. In reality, however, faults in some units may be easier to detect than faults in other units.

In this section, we further extend our models to account for the situation in which the probability of fault detection is differentiated. To deal with this situation, we need to assign different probability values for test cases to detect faults in different units. The challenge in performing such an assignment, however, lies in obtaining effective estimates of the probability of fault detection. In this chapter, we further estimate differentiated p values at the

[2] This limitation is due to the specific algorithm, but conceptually our extended model implementation yields the *total* strategy when $p = 0$.

method level based on two widely used static metrics: *MLoC*, which stands for Method Line of Code, and *McCabe*, which stands for the well-known McCabe Cyclomatic Complexity [87]. Our approach is based on the intuition that methods with larger volume (e.g., higher MLoC values) or greater complexity (e.g., higher McCabe values) need to be covered more times to reveal the faults within them, i.e., they should have lower p values. We believe that test cases should be good at detecting faults, and thus we calculate the p value for each method in the range $[0.5, 1.0]$. Formally, we use both linear normalization (Formula (2.1)) and log normalization (Formula (2.2)) to calculate the p value for the jth method (i.e., $p[j]$) as follows:

$$1 - 0.5 * \frac{Metric[j] - Metric_{min}}{Metric_{max} - Metric_{min}} \tag{2.1}$$

$$1 - 0.5 * \frac{log_{10}(Metric[j] + 1) - log_{10}(Metric_{min} + 1)}{log_{10}(Metric_{max} + 1) - log_{10}(Metric_{min} + 1)} \tag{2.2}$$

where $Metric[j]$ denotes the MLoC or McCabe metric values for the jth method, and $Metric_{min}/Metric_{max}$ denotes the minimum/maximum metric value among all methods of the program under test.[3]

Based on the two metrics and the two p calculation formulas, we thus have four heuristics for generating a differentiated p value for each method. For both models, we change all references to the uniform p into the differentiated $p[j]$ generated for the specific jth method. For the basic model, we change line 36 of Algorithm 1 into $Prob[j] \leftarrow Prob[j] * (1 - p[j])$. Similarly, for

[3]Note that all metric values are increased by 1 in the log normalization to avoid the $log_{10}0$ exception.

the extended model, we change lines 15, 23, and 36 of Algorithm 1 to $sum \leftarrow sum + Prob[j] * (1 - (1 - p[j])^{Cover[k,j]})$, $s \leftarrow s + Prob[j] * (1 - (1 - p[j])^{Cover[l,j]})$, and $Prob[j] \leftarrow Prob[j] * (1 - p[j])^{Cover[k,j]}$, respectively. Note that the worst case time costs of the basic and extended models with differentiated p values are still $O(mn^2)$.

2.4 Implementation

To collect coverage information, we used on-the-fly byte-code instrumentation which dynamically instruments classes loaded into the JVM through a Java agent without any modification of the target program. We implemented code instrumentation based on the *ASM byte-code manipulation and analysis framework* [1]. In particular, we inherited the visitor classes defined in the ASM framework, and added in our code for recording coverage information. To compute Java metrics for each method, we implemented our tool based on the abstract syntax tree (AST) analysis provided by the *Eclipse JDT toolkit* [2]. We extended the Eclipse AST parsing tool to calculate method lines of code (MLoC) and McCabe Cyclomatic complexity (McCabe) metrics.

2.5 Experimental Study

To evaluate our strategies with uniform and differentiated p values in the basic and extended models, we performed an empirical study to investigate the following research questions:

- **RQ1**: How do prioritization strategies generated by the basic and extended models with uniform p values compare with the *total* and *additional* strategies?

- **RQ2**: How do the granularity of coverage and the granularity of test cases impact the comparative effectiveness of strategies generated by our models?

- **RQ3**: How does the use of differentiated p values compare, in terms of effectiveness, with the *total* and *additional* strategies?

2.5.1 Independent Variables

We consider three independent variables:

Prioritization Strategy. We use the following 48 strategies for test-case prioritization. First, as control strategies, we use the *total* and *additional* strategies. Second, for our basic model we use values of p ranging from 0.05 to 0.95 with increments of 0.05, i.e., 19 p values. Third, for our extended model we use the same 19 values of p as those used for our basic model. Fourth, for differentiated p values we use the four p value generation heuristics for both the basic and extended models.

Coverage Granularity. In prior research on test-case prioritization, researchers treated coverage granularity as a constituent part of prioritization techniques. As our aim is to investigate various generic prioritization strategies, we separate coverage granularity from prioritization techniques. As in

prior research, we use structural coverage criteria at both the method level and the statement level. Note that we used differentiated p values only at the method level.

Test-Case Granularity. We consider test-case granularity as an additional factor, at two levels: the test-class level and the test-method level. For the test-class level we treat each JUnit TestCase class as a test case. For the test-method level we treat each test method in a JUnit TestCase class as a test case. That is to say, a test case at the test-class level typically consists of a number of test cases at the test-method level. Section 2.5.3 provides a detailed description.

2.5.2 Dependent Variable

Our dependent variable tracks technique effectiveness. We adopt the well-known APFD (Average Percentage Faults Detected) metric [117]. Let T be a test suite and T' be a permutation of T, the APFD for T' is defined as follows.

$$APFD = \frac{\sum_{i=1}^{n-1} F_i}{n * l} + \frac{1}{2n} \quad (2.3)$$

Here, n is the number of test cases in T, l is the number of faults, and F_i is the number of faults detected by at least one test case among the first i test cases in T'.

2.5.3 Subject Programs, Test Suites, and Faults

As objects of study we consider 19 versions of four programs written in Java, including three versions of *jtopas*, three versions of *xml-security*, five versions of *jmeter*, and eight versions of *ant*. We obtained the programs from the *Software-artifact Infrastructure Repository* (SIR) [7, 35], which provides Java and C programs for controlled experimentation on program analysis and testing. The sizes of the programs range from 1.8 to 80 KLoC. Table 2.1 depicts statistics on the objects. In Table 2.1, for each object program, Columns 3 and 4 present the number of classes (including interfaces) and the number of methods, respectively.

In SIR, each version of each program has a JUnit test suite that was developed during program evolution. Due to the features of JUnit, there are two levels of test-case granularity in these test suites: the test-class level and the test-method level. Column 5 of Table 2.1 depicts the number of all test cases and the number of test cases that detect at least one studied fault for each program at the test-class level. Similarly, Column 6 depicts the test case statistics at the test-method level. As previous research [13, 14, 36] has confirmed that it is suitable to use faults produced via mutation for experimentation in test-case prioritization, we followed a similar procedure to produce faulty versions for each of the 19 object programs. In particular, we used *MuJava* [4, 82] to generate faults and followed the procedure used by Do et al. [36] to select specific mutants to use (as detailed below).

Table 2.1: Statistics on Objects of Study

Object	KLoC	#Class	#Meth	#TClass	#TMeth
jtopas-v1	1.89	19	284	10 (8)	126 (24)
jtopas-v2	2.03	21	302	11 (10)	128 (27)
jtopas-v3	5.36	50	748	18 (8)	209 (25)
xmlsec-v1	18.3	179	1627	15 (3)	92 (10)
xmlsec-v2	19.0	180	1629	15 (1)	94 (7)
xmlsec-v3	16.9	145	1398	13 (8)	84 (41)
jmeter-v1	33.7	334	2919	26 (7)	78 (18)
jmeter-v2	33.1	319	2838	29 (8)	80 (31)
jmeter-v3	37.3	373	3445	33 (16)	78 (43)
jmeter-v4	38.4	380	3536	33 (16)	78 (55)
jmeter-v5	41.1	389	3613	37 (20)	97 (57)
ant-v1	25.8	228	2511	34 (17)	137 (45)
ant-v2	39.7	342	3836	51 (42)	219 (118)
ant-v3	39.8	342	3845	51 (44)	219 (148)
ant-v4	61.9	532	5684	102 (47)	521 (135)
ant-v5	63.5	536	5802	105 (53)	557 (133)
ant-v6	63.6	536	5808	105 (52)	559 (230)
ant-v7	80.4	649	7520	149 (122)	877 (599)
ant-v8	80.4	650	7524	149 (51)	878 (197)

2.5.4 Experiment Procedure

In actual testing scenarios, a specific program version usually does not contain a large number of faults [36]. Therefore, similar to Do et al. [36], we used the mutant pool for each object program to create a set of small mutant groups. To form a mutant group, we randomly selected five mutants that can be killed by one or more test cases in the test suite for the program. For each program, we randomly produced up to 20 mutant groups for each program ensuring that no mutant is used in more than one mutant group. In fact, as there are only 35 mutants of *jmeter-v1* that can be killed by one or more test cases in its test suite, we produced only seven mutant groups for this program. In all other circumstances we produced 20 mutant groups for each program.

Next, we used each of the mutant groups produced for each of the 19 program versions as possible subsequent versions. That is to say, given a

program version V and a generic prioritization strategy S with a coverage-granularity level C_l and a test-case-granularity level T_l, we obtained the effectiveness of strategy S on V for C_l and T_l as follows. First, we used S to obtain a prioritized sequence of test cases for V at C_l and T_l. Then, we calculated the APFD values of the prioritized sequence of test cases for each mutant group of V. These values serve as our data sets for analysis.

2.5.5 Threats to Validity

Our object programs, test cases, and seeded faults may all pose threats to external validity. First, although we used 19 Java program versions of various sizes, the differences seen in our study may be difficult to generalize to other Java programs. Furthermore, our results may not generalize to programs written in languages other than Java. Second, our results based on programs with seeded faults may not be generalizable to programs with real faults. Third, the results may not be generalizable to other test cases. Further reduction of these threats requires additional studies involving additional object programs, test suites, and faults.

The main threat to internal validity for our study is that there may be faults in our implementation of the strategies and the calculation of APFD values. To reduce this threat, we reviewed all the code that we produced for our experiments before conducting the experiments.

To assess technique effectiveness, we used the APFD metric [117] that is widely used for test-case prioritization. However, the APFD metric does

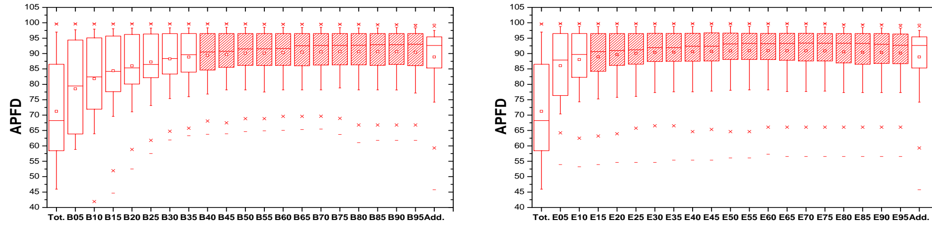


Figure 2.1: Results for test suites at the test-method level with method coverage

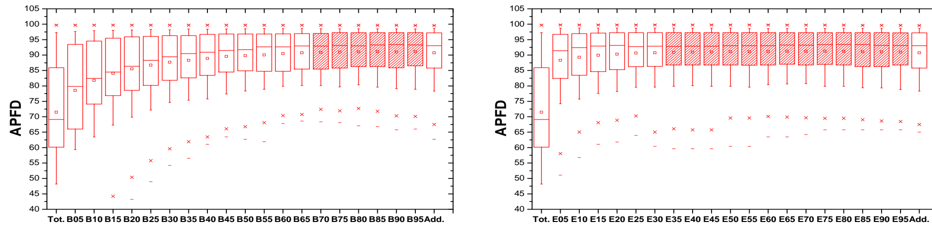


Figure 2.2: Results for test suites at the test-method level with statement coverage

have limitations [36, 117], and we did not consider efficiency or other cost and savings factors. Reducing this threat requires additional studies using more sophisticated cost-benefit models [38].

2.5.6 Results and Analysis

Due to the large number of strategies, various test-case granularities, coverage granularities, objects, and mutant groups studied, box-plots across

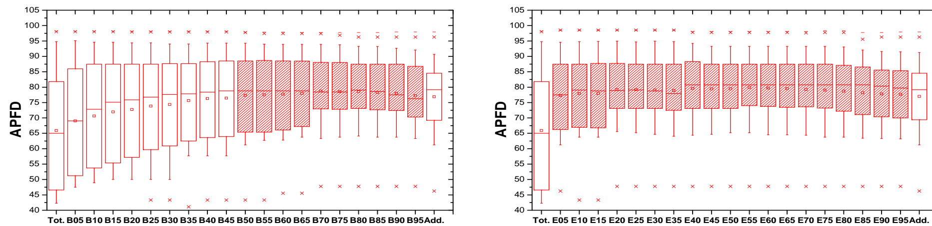


Figure 2.3: Results for test suites at the test-class level with method coverage

Table 2.2: Fisher's LSD test for comparing strategies in the basic model to the *additional* strategy

TCG	CG	$B95$	$B90$	$B85$	$B80$	$B75$	$B70$	$B65$	$B60$	$B55$	$B50$	$B45$	$B40$	$B35$	$B30$	$B25$	$B20$	$B15$	$B10$	$B05$
Test-Meth	Meth	1	1	1	1	1	1	1	0	0	0	0	0	0	0	-1	-1	-1	-1	-1
	Stat	0	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1
Test-Class	Meth	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1
	Stat	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 2.3: Fisher's LSD test for comparing strategies in the extended model to the *additional* strategy

TCG	CG	$E95$	$E90$	$E85$	$E80$	$E75$	$E70$	$E65$	$E60$	$E55$	$E50$	$E45$	$E40$	$E35$	$E30$	$E25$	$E20$	$E15$	$E10$	$E05$
Test-Meth	Meth	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	-1
	Stat	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1
Test-Class	Meth	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
	Stat	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

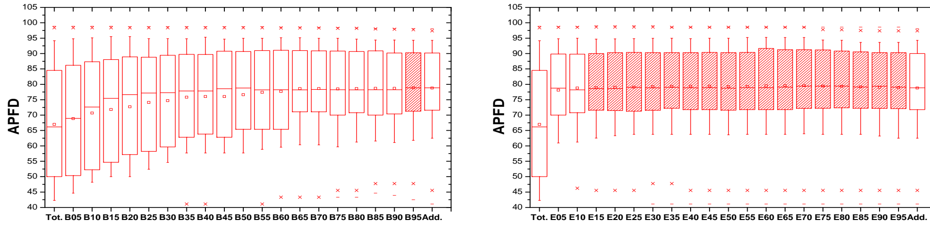


Figure 2.4: Results for test suites at the test-class level with statement coverage

all objects are the most suitable way to show our results.

2.5.6.1 RQ1: Existence of Better Strategies Between the Total and Additional Strategies

Figures 2.1 to 2.4 depict the results of the comparision of the 19 strategies in our basic model and the 19 strategies in our extended model with the *total* and *additional* strategies using test suites at the test-method/test-class level and coverage information at the method/statement level. We denote the *total* strategy as *Tot.* and the *additional* strategy as *Add.*. For a strategy in our basic model, we use *Barbecue* and the value of p to denote the strategy. For example, we use *Barbecue*05 to denote the strategy in our basic model with the p value 0.05. Similarly, for a strategy in our extended model, we use E and the value of p to denote the strategy. Thus, the strategy in our extended model with the value of p set to 0.05 is denoted as $E05$. In each plot, the X-axis shows various strategies compared, and the Y-axis shows the APFD values measured. Each box plot shows the average (dot in the box), median (line in the box), upper/lower quartile, and 90th/10th percentile APFD values

achieved by a strategy over all mutant groups of all 19 versions. For ease of understanding, we mark the strategies with higher average APFD values over the corresponding *additional* strategies as shadowed box plots. Based on the results, we make the following observations.

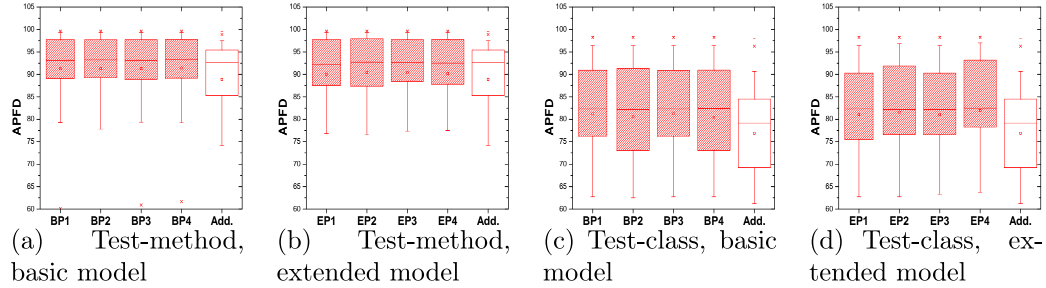
First, when comparing strategies in our approach with the *additional* strategy, strategies with p values between 0.95 and 0.50 in both our basic and extended models typically achieve higher average APFD values. The only exceptions to this are the strategies in our basic model based on statement coverage for test suites at the test-class level with p values between 0.90 and 0.50, and for test suites at the test-method level with p values between 0.65 and 0.50. This observation indicates that there is a wide range of p values that can be used for our models. It should also be noted that the average increases in APFD of our strategies over the *additional* strategy are usually not large. However, considering that the *additional* strategy is widely accepted as the most effective prioritization strategy and is as expensive as our strategies, the increases in APFD are valuable and are actually achieved with almost no extra cost.

Second, when comparing strategies in our approach with the *total* strategy (denoted as *Tot.* in the figures), our strategies with all p values in both our basic and extended models achieve higher average APFD values. One interesting point is that, even when the p value is 0.05 (which results in strategies similar to the *total* strategy), strategies in both our basic and extended models are substantially more effective than the *total* strategy. This observation

indicates that adding a little flavor of the *additional* strategy into the *total* strategy could improve the *total* strategy substantially.

Third, when comparing strategies in our basic model and strategies in our extended model, the strategies perform similarly with p values close to 1 and differently with p values close to 0. When p is close to 1, strategies in both models achieve comparable and even higher APFD values than the *additional* strategy. However, when p is close to 0, strategies in our extended model remain competitive but strategies in our basic model become much less competitive. In other words, strategies with a small p value in our basic model perform more like the *total* strategy, but strategies in our extended model always perform like the *additional* strategy with any p values. In fact, almost all strategies of our extended model with $p \geq 0.15$ outperform the *additional* strategies, except those prioritizing tests at the test-method level using statement coverage with $p \in [0.15, 0.30]$.

As strategies in our models and the *additional* strategy typically achieve similar APFD values, for each coverage-granularity level and each test-case-granularity level, we used Origin [5] to perform a one-way ANOVA analysis of the strategies. The results indicate that there are significant differences among the strategies at the 0.05 significance level. We then used Origin to perform Fisher's LSD test [134] of the strategies. Tables 2.2 and 2.3 (in which TCG stands for test-case granularity, CG stands for coverage granularity, 1 indicates statistically significantly better, 0 indicates no significant difference, and -1 indicates statistically significantly worse) list the results of Fisher's LSD



(a) Test-method, basic model (b) Test-method, extended model (c) Test-class, basic model (d) Test-class, extended model

Figure 2.5: Prioritization results for models embodied with differentiated p values for each method

test for comparing strategies in our models to the *additional* strategy.

According to Tables 2.2 and 2.3, when prioritizing test cases at the test-method level using method coverage, strategies with p values between 0.65 and 0.95 in our basic model and with any p values between 0.30 and 0.95 in our extended model achieve significantly better APFD values than the *additional* strategy. When prioritizing test cases at the test-class level using method coverage, strategies with p values between 0.20 and 0.75 in our extended model significantly outperform the *additional* strategy. Furthermore, the *additional* strategy cannot significantly outperform any strategies in our basic model with p values between 0.50 and 0.95 and any strategies in our extended model with p values between 0.15 and 0.95 in any circumstance. This observation further confirms that our models can achieve clear benefits.

2.5.6.2 RQ2: Impact of Coverage and Test-Case Granularities

Based on Figures 2.1 to 2.4, we make the following observations.

Impact of coverage granularity. Our models seem to be more beneficial when using coverage information at the method level than at the statement level. According to comparisons between Figure 2.1 and Figure 2.2, and between Figure 2.3 and Figure 2.4, in both our basic and extended models, the ranges in which our strategies outperform the *additional* strategy on average are much broader using coverage information at the method level than at the statement level. We suspect the reason for this to be that, when a test case covers a statement, the probability for the test case to detect faults in the statement is very high. Thus, the *additional* strategy is already a good enough strategy for this situation. However, when a test case covers a method, the probability for the test case to detect faults in the covered method is not very high. Thus, we should typically consider that the method may still contain some undetected faults after being covered by some test cases.

Our extended model seems to be applicable for both method coverage and statement coverage. In fact, for all combinations of coverage granularity and test-case granularity, the ranges of strategies in our extended model that outperform the *additional* strategy on average are all very wide (i.e., for any $p > 0.30$).

As our empirical results indicate that our models are more beneficial with method coverage, we further compare our strategies using method coverage with the *additional* strategy using statement coverage. When prioritizing test cases at the test-method level, the average APFD values of wide ranges of strategies in our models (i.e., strategies in the basic model with p values

between 0.75 and 0.90, and strategies in the extended model with p values between 0.45 and 0.75) using method coverage are very close to the average APFD values of the *additional* strategy using statement coverage. When prioritizing test cases at the test-class level, the average APFD values of wide ranges of strategies in our models (i.e., strategies in the basic model with p values between 0.70 and 0.80, and strategies in the extended model with p values between 0.20 and 0.80) using method coverage are as competitive as or even better than the average APFD values of the *additional* strategy using statement coverage. We also performed an ANOVA analysis (at the 0.05 level) and Fisher's LSD test to compare our strategies and the *additional* strategy using method coverage to the *additional* strategy using statement coverage. The ANOVA analysis and Fisher's LSD test demonstrate that there is no statistically significant difference between the *additional* strategy using statement coverage and any strategy in our basic model with any p value between 0.50 and 0.95 or any strategy in our extended model with any p value between 0.20 and 0.95 using method coverage. However, the *additional* strategy using statement coverage is significantly better than the *additional* strategy using method coverage. As coverage information at the method level is usually much less expensive to acquire than coverage information at the statement level, this result indicates that wide ranges of strategies in our models using method coverage can serve as cheap alternatives for the *additional* strategy using statement coverage.

Impact of test-case granularity. Our extended model seems to be more

beneficial than our basic model for prioritizing test cases at the test-class level. First, when using the extended model instead of the basic model, the number of strategies that outperform the *additional* strategies increases more dramatically at the test-class level than at the test-method level (shown in Figures 2.1 to 2.4). Second, when prioritizing test cases at the test-method level, the largest average APFD values achieved by our extended model are larger than those achieved by our basic model by 0.15 (statement coverage) and 0.25 (method coverage), respectively. However, when prioritizing test cases at the test-class level, the differences are 0.69 (statement coverage) and 1.03 (method coverage), respectively. Third, results of our statistical analysis shown in Tables 2.2 and 2.3 also confirm this observation. We suspect the reason for this to be that it is more common for a test case at the test-class level than a test case at the test-method level to cover a method or a statement more than once. In such a circumstance, it is more beneficial to consider multiple coverage information.

All the strategies that we considered achieve significantly higher average APFD values for prioritizing test cases at the test-method level than for prioritizing test cases at the test-class level. In fact, for any object and strategy, using either statement coverage or method coverage, the average APFD value for prioritizing test cases at the test-method level is uniformly higher than that for prioritizing test cases at the test-class level. We suspect the reason for this to be that, as a test case at the test-class level consists of a number of test cases at the test-method level, it is more flexible to prioritize

test cases at the test-method level.

Table 2.4: Fisher's LSD test for comparing p-differentiated techniques with p-uniform techniques at the test-method level

Tech.	$BP1$	$BP2$	$BP3$	$BP4$	$EP1$	$EP2$	$EP3$	$EP4$
M-$B75$	0	0	0	0	0	0	0	0
M-$E60$	0	0	0	0	0	0	0	0
M-*Add.*	1	1	1	1	1	1	1	1
S-$B85$	0	0	0	0	0	0	0	0
S-$E65$	0	0	0	0	0	0	0	0
S-*Add.*	0	0	0	0	0	0	0	0

Table 2.5: Fisher's LSD test for comparing p-differentiated techniques with p-uniform techniques at the test-class level

Tech.	$BP1$	$BP2$	$BP3$	$BP4$	$EP1$	$EP2$	$EP3$	$EP4$
M-$B70$	1	0	1	0	1	1	1	1
M-$E55$	0	0	0	0	0	0	0	1
M-*Add.*	1	1	1	1	1	1	1	1
S-$B95$	1	0	1	0	1	1	1	1
S-$E70$	0	0	0	0	0	1	0	1
S-*Add.*	1	0	1	0	1	1	1	1

2.5.6.3 RQ3: Using Differentiated p Values

Figure 2.5 depicts results obtained by comparing the p-differentiated strategies with the corresponding *additional* strategies. We use BP to denote the four strategies in the basic model, and EP to denote the four strategies in the extended model. For the basic model, $BP1$ denotes the use of the MLoC metric and linear normalization, $BP2$ denotes the use of the MLoC metric and log normalization, $BP3$ denotes the use of the McCabe metric and linear normalization, and $BP4$ denotes the use of the McCabe metric and log normalization. The naming of strategies in the extended model follows the same manner. In the box plots, the X-axis denotes the studied strategies, the Y-axis denotes the APFD values achieved by compared strategies, and

each box denotes the results of a strategy on all mutant groups of all objects. We also performed an ANOVA analysis (at the 0.05 level) and Fisher's LSD test to compare the eight strategies with differentiated p values to the best strategies in our basic/extended models and *additional* strategies using method and statement coverage. Tables 2.4 and 2.5 show the Fisher's LSD test result, where "M-" denotes the strategies using method coverage, and "S-" denotes the strategies using statement coverage. For example, "M- *Barbecue*70" denote the *Barbecue*70 strategy using method coverage. We make the following observations.

First, all strategies with differentiated p values outperform the corresponding *additional* strategies based on method coverage substantially. Figure 2.5 shows that all the strategies with differentiated p values achieve higher APFD values over corresponding addtional strategies on average. For example, when prioritizing test-class-level tests using method coverage, the *additional* strategy achieves an APFD value of 76.88 on average, while the four strategies from the extended model achieve APFD values from 81.10 to 81.92. In addition, Table 2.4 shows that all eight strategies are statistically significantly better than the *additional* strategy based on method coverage under test-method granularity, and Table 2.5 shows that all eight strategies are statistically significantly better than the *additional* strategy based on method coverage under test-class granularity.

Second, all strategies with differentiated p values using method coverage are comparable to the best strategies in our basic and extended models

(including strategies using method and statement coverage) and the *additional* strategies using statement coverage, and even outperform some of those techniques. At both test-class and test-method granularities, the eight strategies are not statistically inferior to any best strategies within our basic/extended models or *additional* strategies using statement coverage. At the test-class granularity, six of the eight strategies are statistically significantly better than the *additional* strategy using statement coverage and the best strategies of the basic model using method coverage and statement coverage. This indicates that strategies with differentiated p values using method coverage can even be a *cheaper but better* alternative choice for prioritization techniques using statement coverage.

2.5.7 Implications

Here are the main findings of our experimental study:

- For a wide range of p values (i.e., between 0.95 and 0.50), strategies in both our basic and extended models (on average) outperform or are at least competitive with the *additional* strategy using any combination of test-case and coverage granularities.

- Strategies in the extended model are generally more effective than strategies in the basic model, especially when the values of p are close to 0.

- Strategies in the basic and extended models are more beneficial for method coverage than statement coverage.

- Our extended model is more beneficial for test suites at the test-class level, while our basic model is more suitable for test suites at the test-method level.

- All our strategies using differentiated p values statistically significantly outperform the *additional* strategies using method coverage. Some of our strategies using differentiated p values with method coverage even statistically significantly outperform the *additional* strategies using statement coverage.

The experimental findings provide implications for practitioners. The need for more and better blended approaches provides implications for researchers.

2.6 Summary

The main contributions of this chapter are as follows.

- A new approach that creates better prioritization techniques by controlling the uncertainty of fault detection capability in test-case prioritization.

- Two models that unify the *total* and *additional* strategies and can also yield a spectrum of generic strategies having flavors of both the *total* and *additional* strategies.

- Empirical evidence that many strategies between the *total* and *additional* strategies are more effective than either of those strategies.

- Empirical evidence that our strategies using differentiated p values with method coverage can significantly outperform the *additional* strategy with statement coverage.

Chapter 3

Selective Mutation Testing

The previous chapter presented an approach that I introduced in the regression testing area. This chapter presents my approach for more efficient selective mutation testing in the mutation testing area, which was presented at the 28th IEEE/ACM Conference on Automated Software Engineering (ASE 2013) [147].[1]

3.1 Background

While mutation testing [16,32,43,46,92,96,121,149,155] could be useful in many domains, it is extremely expensive. For example, a mutation testing tool for C, Proteum [30], implements 108 mutation operators that generate 4,937 mutants for a small C program with only 137 non-blank, non-comment lines of code [149]. Therefore, generating and (especially) executing the large number of mutants against the test suite under evaluation is costly. Various methodologies for reducing the cost of mutation testing have been proposed. One widely used methodology is *selective mutation testing* [12,16,43,92,96,101,

[1]Note that this is a joint work [147] with another PhD student, Milos Gligoric, from University of Illinois at Urbana-Champaign.

121, 149], which only generates and executes a subset of mutants for mutation testing. Ideally, the selected subset of mutants should be representative of the entire set of mutants.

The most widely studied approach for selective mutation testing is *operator-based mutant selection* [16, 43, 92, 96, 101, 149], which only generates mutants using a subset of mutation operators; the selected subset of mutation operators is required to be *effective*, i.e., if a test suite kills all the non-equivalent mutants generated by the selected set of operators (i.e., the test suite is *adequate* for selected mutants), then the test suite should kill (almost) all the non-equivalent mutants generated by all mutation operators. Further, selected operators should lead to high savings; the *savings* is usually calculated as the ratio of non-selected mutants over all the mutants. Researchers also evaluated a simple approach of *random mutant selection* [43,136,149], and a recent study [149] reported that random selection is as effective as operator-based mutant selection when random selection selects the same number of mutants from *all mutants* as the operator-based selection selects.

Although the existing approaches are effective, mutation testing remains one of the most expensive methodologies in software testing. No previous study has explored how to further reduce the number of mutants generated by operator-based mutant selection, and whether operator-based selection and random selection can be combined. Also, previous work has not explored how random mutant selection and operator-based selection relate for test suites that do not kill all non-equivalent mutants (i.e., *non-adequate* test suites). In

addition, all the studies for sequential mutants [16, 92, 96, 101, 136, 149] evaluated mutant selection on small C and Fortran programs—the largest program used for selective mutation testing was only 513 lines of code. Empirical studies on larger, real-world programs are lacking.

In this chapter, we investigate a simple idea, called *sampling mutation*, that applies random selection on the set of mutants generated by operator-based selection (rather than on the set of mutants generated by all operators [43, 136, 149]); we call the process of obtaining the mutants *sampling*, the percentage of randomly selected mutants the *sampling ratio*, and the resulting set of mutants a *sample*. We introduce new sampling strategies that select mutants based on the program elements not only based on the mutation operators. Additionally, we report an extensive empirical evaluation on 11 real-world Java projects of various sizes to show that sampling mutation remains effective and has a high predictive power, even with high savings. Interestingly, for all the subjects, we find that sampling only 5% of the mutants generated by operator-based mutant selection is effective and has a high correlation with results on mutants selected by operators while having 20x fewer mutants to execute.

3.2 Study Approach

3.2.1 Problem Definition

Given a program under test, P, and a test suite, T, we denote the set of all *selected mutants* generated by operator-based mutant selection as

M, and the set of non-equivalent mutants in M as NEM. Following existing studies [16, 92, 96, 149], we randomly construct n test suites of various sizes $\{T_1, T_2, ..., T_n\}$; the set of mutants that can be killed by T_i ($1 \leq i \leq n$) is denoted $K(T_i, M)$. Then the (actual) *selected mutation score* achieved by T_i over the selected mutants M is defined as:

$$MS(T_i, M) = \frac{|K(T_i, M)|}{|NEM|} \tag{3.1}$$

In this study, we apply a set of sampling strategies on top of the selected mutants. Let S be a sampling strategy; the set of *mutants sampled* by S from M is denoted M_S. We apply each strategy m times (with different random seeds) to generate a set of mutant samples: $\{M_{S_1}, M_{S_2}, ..., M_{S_m}\}$. The set of mutants in M_{S_j} ($1 \leq j \leq m$) that are killed by test suite T_i ($1 \leq i \leq n$) is denoted $K(T_i, M_{S_j})$. Then, the *sampling mutation score* achieved by T_i over M_{S_j} can be represented as:

$$MS(T_i, M_{S_j}) = \frac{|K(T_i, M_{S_j})|}{|M_{S_j} \cap NEM|} \tag{3.2}$$

Intuitively, if $MS(T_i, M_{S_j})$ is close to $MS(T_i, M)$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$, we say that the sampling strategy S applied on top of selected mutants is *effective* at predicting the result that would be obtained on all selected mutants. (Section 3.3 precisely defines the predictive power.)

3.2.2 Measurement

In the literature, there are two main approaches for evaluating the effectiveness of how a subset of mutants represents a larger set of mutants.

(Traditionally, the sets are generated by all operators, and the subsets are selected mutants; in our study, the sets are selected mutants, and the subsets are sampled mutants.) First, researchers [16, 96, 101, 149] construct test suite T_i $(1 \leq i \leq n)$ that *can* kill all non-equivalent mutants from the subset (called *adequate* test suites), and calculate the mutation score of T_i on the original set of mutants. Second, Namin et al. [92] also examined how, for test suites T_i $(1 \leq i \leq n)$ that *cannot* kill all the non-equivalent mutants from the subset (called *non-adequate* test suites), the mutation score of T_i on the subset of mutants compares with the mutation score of T_i on the original set of mutants.

In this study, we use both approaches to evaluate the sampling strategies applied on top of operator-based mutant selection. For the first approach, since our sampling strategy may select different subsets of mutants at different runs, we randomly construct n adequate test suites that can kill all sampled non-equivalent mutants for each of the m sampling runs. We denote the ith $(1 \leq i \leq n)$ test suite that kills all sampled non-equivalent mutants in the jth $(1 \leq j \leq m)$ run of sampling (i.e., the selected mutants are M_{S_j}) as T_{ij}. Following previous work [149], we use the following formula to measure the *effectiveness* of a sampling technique S:

$$EF(S) = \frac{\Sigma_{j=1}^{m} \Sigma_{i=1}^{n} |MS(T_{ij}, M)|}{n * m} \times 100\% \tag{3.3}$$

The only difference between our formula and the original formula [149] is that we also average over m sampling runs; the previous work did not average over different runs because each run of their operator-based selection gives a fixed

subset of mutants. Also note that in the evaluation, we present the standard deviation (SD) values across m sampling runs to show the *stability* of the sampling strategies.

For the second approach, we randomly construct k non-adequate test suites ($\{T_1, T_2, ..., T_k\}$) for each subject and check how m runs of sampling influence the mutation scores of the constructed test suites. We use the correlation between sampling mutation score of T_i and selected mutation score of T_i to measure the *predictive power* of a random sampling strategy S over the mutants generated by operator-based mutant selection:

$$PP(S) = Corr(\{\langle MS(T_i, M_{S_j}), MS(T_i, M)\rangle \\ |1 \leq i \leq k \wedge 1 \leq j \leq m\}) \quad (3.4)$$

The correlation analysis is between the mutation scores on the sampled mutant set M_{S_j} and the mutation scores on the selected mutants M for all constructed test suites for all sampling runs ($1 \leq j \leq m$). Section 3.3 presents, both visually and statistically, the $Corr$ functions we use. To illustrate, for all the m sampling runs of a strategy S, if we use the mutation scores of the k tests suites on sampled mutants M_{S_j} ($1 \leq j \leq m$), i.e., $MS(T_i, M_{S_j})$ ($1 \leq i \leq k$) as the x-axis values, for each x value we will have a corresponding y value to predict, which is $MS(T_i, M)$ ($1 \leq i \leq k$). For a perfect strategy, the graph will be the straight line function $y = x$, which means all the k test suites have exactly the same mutation score on sampled mutants and all original mutants before sampling.

3.2.3 Combining Operator-Based and Random Mutant Selection

Given selected mutants, M, we define eight random sampling strategies that specify which mutants to select from M.

- **Baseline Strategy**, which samples $x\%$ mutants from the selected set of mutants M. Formally, the set of mutants sampled by strategy S_{base} can be defined as:

$$M_{S_{base}} = Sample(M, x\%)$$

  where $Sample(M, x\%)$ denotes random sampling of $x\%$ mutants from M.[2]

- **MOp-Based Strategy**, which samples $x\%$ mutants from each set of mutants generated by the same mutation operator. Assume the sets of mutants generated by the set of selective mutation operators, say $op_1, op_2, ..., op_k$, are M_{op_1}, M_{op_2}, ..., M_{op_k}, i.e., $M = \cup_{i=1}^{k} M_{op_i}$. Then, the set of mutants sampled by strategy S_{mop} can be formally defined as:

$$M_{S_{mop}} = \cup_{i=1}^{k} Sample(M_{op_i}, x\%)$$

- **PElem-Based Strategies**, which sample $x\%$ mutants from each set of mutants generated inside the same program element (e.g., class, method, or statement). Assume the sets of mutants generated for the set of

[2] If the sample size is a float f, we first sample $\lfloor f \rfloor$ mutants at random, and then with probability $f - \lfloor f \rfloor$ pick one more mutant at random.

elements in the project under test are M_{e_1}, M_{e_2}, ..., M_{e_k}, i.e., $M = \cup_{i=1}^{k} M_{e_i}$. Then, the set of sampled mutants can be defined as:

$$M_{S_{pelem}} = \cup_{i=1}^{k} Sample(M_{e_i}, x\%)$$

In this way, S_{class}, S_{meth}, and S_{stmt} can be defined when using the program element granularities of class, method, and statement, respectively.

- **PElem-MOp-Based Strategies**, which sample $x\%$ mutants from each set of mutants generated by the same mutation operator inside the same program element. Assume the sets of mutants generated for the set of program elements in the project under test are M_{e_1}, M_{e_2}, ..., M_{e_k}, then $M = \cup_{i=1}^{k} M_{e_i}$. Also assume the sets of mutants generated by the set of selective mutation operator are M_{op_1}, M_{op_2}, ..., M_{op_h}, then $M = \cup_{j=1}^{h} M_{op_j}$. Finally, the set of sampled mutants can be defined as:

  $$M_{S_{pelem-mop}} = \cup_{i=1}^{k} \cup_{j=1}^{h} Sample(M_{e_i} \cap M_{op_j}, x\%)$$

  In this way, $S_{class-mop}$, $S_{meth-mop}$, and $S_{stmt-mop}$ can be defined when using the program element granularities of class, method, and statement, respectively.

Note that the first two strategies, S_{base} and S_{mop}, have been used by previous studies [136, 149] to evaluate random mutant selection from *all mutants*. We believe that using all mutants as the candidate set may be unnecessary. Therefore, we use these two strategies to evaluate random mutant sampling

Table 3.1: Subject programs used in the evaluation

Subject	LOC	#Tests	#Mutants	
			All	Killed
TimeMoney^{r207} [130]	2681	236	2304	1667
JDepend$^{v2.9}$ [56]	2721	55	1173	798
JTopas$^{v2.0}$ [68]	2901	128	1921	1103
Barbecue^{r87} [15]	5391	154	36418	1002
Mime4J$^{v0.50}$ [90]	6954	120	19111	4414
Jaxen^{r1346} [55]	13946	690	9880	4616
XStream$^{v1.41}$ [139]	18369	1200	18046	10022
XmlSecurity$^{v3.0}$ [138]	19796	83	9693	2560
CommonsLang^{r1040879} [25]	23355	1691	19746	12970
JodaTime^{r1604} [65]	32892	3818	24174	16063
JMeter$^{v1.0}$ [63]	36910	60	21896	2024

from *operator-based selected mutants*. In addition, our three S_{pelem} strategies, which aim to sample mutants across all program locations evenly, are *the first* to randomly sample mutants at the program element dimension. Furthermore, our three $S_{pelem-mop}$ strategies are *the first* to sample mutants across two dimensions: mutation operators and program elements.

3.3 Empirical Study

We performed an extensive empirical evaluation to demonstrate the effectiveness, predictive power, and savings of the proposed sampling strategies.

3.3.1 Subject Programs

The evaluation includes a broad set of Java programs from various sources. We chose programs of different sizes (from 2681 to 36910 LOC) to explore the benefits of our sampling strategies for various cases.

Table 3.1 shows 11 subject programs used in the evaluation: TimeMoney, a set of classes for manipulating time and money; JDepend, a tool for measuring the quality of code design; JTopas, a library for parsing arbitrary text data; Barbecue, a library for creating barcodes; Mime4J, a parser for e-mail message streams in MIME format; Jaxen, an implementation of XPath engine; XStream, a library for fast serialization/deserialization to/from XML; XmlSecurity, an Apache project that implements security standards for XML; CommonsLang, an Apache project that extends standard Java library; Joda-Time, a replacement for standard Java date and time classes; and JMeter, an Apache project for performance testing. All the 11 subjects have been widely used in software testing research [121, 122, 148, 153, 155, 161].

Table 3.1 includes some characteristics of the programs. Column "Subject" shows the name of each subject program, the version/revision number (as applicable) and the reference to the webpage with sources; "LOC" shows the number of non-blank lines of code measured by JavaSourceMetric [64]; "#Tests" shows the number of available tests for the program (it is important to note that we have not created any special test for the purpose of this study: all the tests for 11 subjects come from their code repositories, and to the best of our knowledge, all these tests are manually written); "#MutantsAll" and "#MutantsKilled" show the total number of mutants that Javalanche [121,122] generated using operator-based selection and the number of killed mutants, respectively. We used Javalanche because it is a state-of-the-art mutation testing tool for Java programs. It generates mutants using the operator-based

mutant selection approach proposed by Offutt et al. [96, 101]. Specifically, Javalanche uses the following four mutation operators: Negate Jump Condition, Omit Method Call, Replace Arithmetic Operator, and Replace Numerical Constant. Note that the subjects used in our study are orders of magnitude larger than the subjects used in previous studies on selective mutation testing [92, 96, 101, 136, 149].

3.3.2 Experimental Design

We next describe our experimental setup and the data we collected.

3.3.2.1 Independent Variables

We used the following independent variables in the study:

IV1: Different Random Sampling Strategies. We apply each of our eight sampling strategies on top of the mutants generated by operator-based mutant selection, to investigate their effectiveness, predictive power, and savings.

IV2: Different Sampling Ratios. For each sampling strategy S, we use 19 sampling ratios $r \in \{5\%, 10\%, ..., 95\%\}$.

IV3: Different Subject Sizes. For each strategy S with each ratio r, we apply S on all the subjects with various sizes, and investigate the differences.

3.3.2.2 Dependent Variables

We used the following dependent variables to investigate the output of the experiments:

DV1: Effectiveness. For the mutants sampled by each strategy S among all selected mutants, we construct test suites that can kill all sampled non-equivalent mutants, and record the selected mutation score of those test suites. The higher the selected mutation score is, the more effective the selected mutants are for evaluating test suites (Equation 3.3). (The same experimental procedure was used previously to measure the effectiveness of operator-base selection and random selection [16, 43, 92, 96, 101, 149].)

DV2: Predictive Power. For each sampling strategy S, we also construct test suites that do not kill all sampled non-equivalent mutants, and use statistical analysis to measure the predictive power of the sampled mutants (equation 3.4). If the constructed test suites have similar values for sampling mutation score and selected mutation score, then the sampled mutants are a good predictor of the selected mutants. More precisely, we instantiate the $Corr$ function to measure: R^2 coefficient of determination for linear regression, Kendall's τ rank correlation coefficient, and Spearman's ρ rank correlation coefficient.

DV3: Time Savings. For each triple (P, S, r) of subject program P, sampling strategy S, and sampling ratio r, we compare the mutation testing time for the sampled mutants and the mutation testing time for the selected mutants.

3.3.2.3 Experimental Setup

Following previous studies on selective mutation testing [92, 149], we deemed all mutants that cannot be killed by any test from the original test

suite as equivalent mutants in our study. We evaluate all the sampling strategies with all sampling ratios on all subjects. Given a subject program and selected mutants for that program, we first run sampling 20 times for each of 8 sampling strategies with each of the 19 sampling ratios. As a result, we get 20*8*19=3,040 samples of mutants for each subject program.

Then, for each sample of mutants, we randomly construct 20 adequate test suites that each kill all the non-equivalent mutants in sampled mutants, i.e., we construct 20*3,040=60,800 test suites for each subject. Next, we measure the selected mutation score for each test suite. Each test suite is randomly constructed by including one test at a time until all sampled non-equivalent mutants are killed. We deviate from the previous work [96, 101, 149] that constructed test suites by including multiple tests at a time (using increment of 50 or 200), as such decision can lead to large test suites and high selected mutation scores that do not correspond to practice. By including one test at a time, we simulate a more realistic use of mutation testing in practice, where a user could include one test at a time until all the mutants are killed.

Next, for each subject, we randomly construct 100 (non-adequate) test suites of various sizes that do not necessary kill all the sampled mutants. We randomly construct each test suite by uniformly selecting the size of the test suite to be between 1 and the number of tests available for the subject. Note that our experiments differ in this step from previous work [92], where 100 test suites were generated by taking two test suites for each size between 1 and 50. The reason to deviate from previous work is that our programs greatly differ

in size and number of tests, which was not the case in previous studies. For example, taking sizes between 1 and 50 does not seem appropriate for both Barbecue and JodaTime (with 154 and 3818 tests, respectively). Therefore, we uniformly select the sizes of the test suites up to the total number of tests for each subject program. Then we measure the sampling mutation score (i.e., the mutation score on the sampled mutants) and selected mutation score (i.e., the mutation score on the selected mutants) achieved by each of the constructed test suites. We further perform correlation analysis between the sampling mutation score and the selected mutation score for all test suites on each strategy and ratio combination on each subject. (Section 3.3.3.2 shows the details.)

Finally, for each sample of mutants, we also trace the time for generating and executing the mutants. Although it is common in the literature to report the savings in terms of the number of mutants not generated, this information is implicitly given in our study through the sampling ratio (e.g., if a sampling ratio is 5%, we have 20x fewer mutants). Therefore, our study also reports the mutation execution time in order to confirm that savings in terms of the number of mutants correspond to the savings in terms of mutation execution time for mutation sampling. We performed all experiments on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

3.3.3 Results and Analysis

We report the most interesting findings of our study in this section, while some additional results and detailed experimental data are publicly available online [119].

3.3.3.1 Effectiveness for Adequate Test Suites

Table 3.2 shows the selected mutation scores achieved by randomly constructed adequate test suites that achieve 100% sampled mutation score, i.e., kill all the sampled non-equivalent mutants. According to our experimental setup, for each triple of subject program, strategy, and sampling ratio, (P, S, r), we obtain 20 samples of mutants and construct 20 adequate test suites for each sample. Thus, for each (P, S, r), we show the average selected mutation score and standard deviation achieved by the 20*20=400 test suites. Specifically, column "Ra." shows sampling ratio, column "Subject" shows the subject name, and columns 3-18 show the average values and standard deviations achieved by 8 sampling strategies. The results for all the 19 sampling ratios can be found on the project webpage [119]. Based on the obtained values, we make several observations as follows.

First, for all subjects and all sampling strategies, one can see that the sampled mutants are extremely effective, i.e., the sampled mutants are representative of the selected mutants. For example, even when sampling 5% of the selected mutants, the test suites that kill all the sampled mutants can kill almost all selected mutants. To illustrate, when sampling 5% of selected

Table 3.2: Selected mutation scores (%) achieved by the test suites that achieve 100% sampled mutation scores

Ra.	Subjects	Base		MOp		Class		Meth		Stmt		Class-MOp		Meth-MOp		Stmt-MOp	
		MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.
5%	TimeMoney	99.14	1.12	99.30	0.98	99.15	1.20	99.32	0.90	99.35	0.94	99.10	1.23	99.26	1.00	99.09	1.17
	JDepend	98.23	1.85	98.10	2.09	97.81	2.34	98.31	1.91	98.31	1.71	97.99	2.31	98.54	1.84	98.67	1.54
	JTopas	99.37	1.09	99.24	1.23	99.32	1.18	99.25	1.18	99.30	1.22	99.19	1.35	99.25	1.31	99.16	1.30
	Barbecue	98.63	1.66	98.64	1.82	98.92	1.56	98.99	1.26	99.03	1.37	98.64	1.76	99.04	1.41	98.69	1.73
	Mime4J	99.77	0.34	99.78	0.34	99.77	0.32	99.81	0.29	99.76	0.32	99.82	0.26	99.80	0.28	99.78	0.39
	Jaxen	99.72	0.33	99.69	0.36	99.67	0.37	99.69	0.33	99.67	0.42	99.70	0.36	99.64	0.39	99.70	0.36
	XStream	99.85	0.21	99.86	0.17	99.87	0.18	99.90	0.14	99.88	0.15	99.87	0.18	99.88	0.15	99.85	0.19
	XmlSecurity	99.62	0.61	99.53	0.74	99.68	0.56	99.69	0.50	99.64	0.70	99.73	0.46	99.73	0.45	99.68	0.56
	CommonsLang	99.90	0.13	99.89	0.15	99.89	0.14	99.92	0.10	99.90	0.14	99.89	0.14	99.90	0.12	99.89	0.13
	JodaTime	99.91	0.12	99.91	0.11	99.91	0.11	99.92	0.09	99.92	0.11	99.91	0.11	99.91	0.10	99.91	0.12
	JMeter	99.71	0.53	99.73	0.52	99.76	0.43	99.76	0.37	99.72	0.48	99.72	0.51	99.77	0.44	99.67	0.59
	Avg.	**99.44**	-	**99.42**	-	**99.43**	-	**99.51**	-	**99.50**	-	**99.41**	-	**99.52**	-	**99.46**	-
10%	TimeMoney	99.61	0.61	99.66	0.47	99.71	0.43	99.75	0.37	99.67	0.48	99.67	0.48	99.69	0.45	99.72	0.43
	JDepend	99.21	1.17	99.27	1.03	99.35	0.90	99.40	0.89	99.43	0.84	99.32	0.92	99.38	0.91	99.32	0.87
	JTopas	99.67	0.61	99.62	0.65	99.74	0.44	99.66	0.54	99.65	0.65	99.64	0.61	99.76	0.46	99.75	0.43
	Barbecue	99.45	0.79	99.36	0.95	99.42	0.82	99.60	0.64	99.56	0.62	99.54	0.72	99.47	0.88	99.49	0.74
	Mime4J	99.91	0.17	99.90	0.16	99.93	0.13	99.91	0.15	99.91	0.15	99.92	0.13	99.92	0.13	99.92	0.14
	Jaxen	99.85	0.19	99.85	0.20	99.87	0.17	99.87	0.17	99.87	0.17	99.85	0.19	99.86	0.18	99.87	0.16
	XStream	99.95	0.08	99.94	0.08	99.94	0.08	99.96	0.06	99.96	0.07	99.95	0.09	99.95	0.07	99.95	0.09
	XmlSecurity	99.88	0.23	99.86	0.27	99.88	0.22	99.86	0.23	99.88	0.20	99.86	0.26	99.86	0.25	99.88	0.25
	CommonsLang	99.96	0.06	99.96	0.06	99.96	0.06	99.97	0.04	99.96	0.07	99.95	0.06	99.96	0.06	99.96	0.06
	JodaTime	99.96	0.05	99.96	0.05	99.96	0.06	99.97	0.04	99.97	0.04	99.96	0.06	99.97	0.05	99.97	0.05
	JMeter	99.86	0.27	99.87	0.25	99.90	0.21	99.92	0.17	99.88	0.23	99.88	0.27	99.89	0.24	99.88	0.25
	Avg.	**99.76**	-	**99.75**	-	**99.79**	-	**99.81**	-	**99.79**	-	**99.78**	-	**99.79**	-	**99.79**	-
15%	TimeMoney	99.81	0.30	99.81	0.30	99.82	0.29	99.88	0.19	99.86	0.23	99.84	0.27	99.86	0.24	99.81	0.30
	JDepend	99.42	0.84	99.50	0.82	99.69	0.55	99.63	0.53	99.74	0.48	99.63	0.57	99.59	0.59	99.62	0.60
	JTopas	99.82	0.32	99.82	0.35	99.80	0.32	99.84	0.28	99.87	0.24	99.85	0.26	99.83	0.32	99.79	0.37
	Barbecue	99.68	0.50	99.73	0.40	99.70	0.47	99.74	0.36	99.70	0.46	99.72	0.47	99.74	0.41	99.69	0.52
	Mime4J	99.95	0.10	99.95	0.10	99.96	0.08	99.96	0.09	99.96	0.08	99.94	0.10	99.97	0.08	99.95	0.10
	Jaxen	99.90	0.15	99.91	0.12	99.92	0.11	99.92	0.10	99.92	0.12	99.92	0.10	99.92	0.12	99.92	0.11
	XStream	99.97	0.05	99.97	0.05	99.97	0.06	99.98	0.04	99.98	0.04	99.97	0.05	99.97	0.04	99.97	0.06
	XmlSecurity	99.91	0.19	99.94	0.12	99.93	0.16	99.92	0.15	99.94	0.11	99.92	0.14	99.95	0.11	99.93	0.15
	CommonsLang	99.97	0.05	99.98	0.04	99.97	0.04	99.99	0.02	99.98	0.03	99.98	0.03	99.98	0.03	99.98	0.04
	JodaTime	99.98	0.03	99.98	0.03	99.98	0.03	99.99	0.02	99.98	0.02	99.98	0.03	99.98	0.03	99.98	0.03
	JMeter	99.93	0.16	99.93	0.17	99.93	0.18	99.96	0.12	99.94	0.15	99.94	0.17	99.96	0.14	99.94	0.15
	Avg.	**99.85**	-	**99.87**	-	**99.88**	-	**99.89**	-	**99.90**	-	**99.88**	-	**99.89**	-	**99.87**	-
20%	TimeMoney	99.89	0.19	99.88	0.21	99.88	0.20	99.91	0.15	99.91	0.16	99.88	0.20	99.90	0.21	99.89	0.18
	JDepend	99.73	0.45	99.74	0.45	99.71	0.52	99.78	0.38	99.83	0.33	99.80	0.34	99.78	0.41	99.73	0.48
	JTopas	99.89	0.20	99.89	0.20	99.86	0.26	99.90	0.20	99.89	0.20	99.89	0.22	99.85	0.31	99.87	0.27
	Barbecue	99.80	0.38	99.72	0.44	99.81	0.31	99.84	0.28	99.82	0.31	99.80	0.33	99.81	0.33	99.78	0.39
	Mime4J	99.97	0.07	99.97	0.07	99.98	0.05	99.97	0.05	99.98	0.04	99.97	0.06	99.97	0.06	99.97	0.07
	Jaxen	99.95	0.07	99.94	0.09	99.94	0.09	99.95	0.08	99.94	0.08	99.95	0.07	99.94	0.08	99.94	0.09
	XStream	99.98	0.03	99.98	0.04	99.98	0.03	99.99	0.02	99.98	0.03	99.98	0.03	99.99	0.03	99.98	0.03
	XmlSecurity	99.95	0.09	99.94	0.13	99.95	0.10	99.96	0.08	99.97	0.07	99.94	0.15	99.95	0.10	99.95	0.10
	CommonsLang	99.99	0.02	99.98	0.03	99.99	0.02	99.99	0.02	99.99	0.02	99.98	0.03	99.99	0.02	99.99	0.03
	JodaTime	99.99	0.02	99.99	0.02	99.98	0.02	99.99	0.01	99.99	0.01	99.99	0.02	99.99	0.02	99.99	0.02
	JMeter	99.95	0.15	99.95	0.13	99.95	0.14	99.98	0.09	99.96	0.12	99.97	0.10	99.98	0.07	99.94	0.16
	Avg.	**99.92**	-	**99.91**	-	**99.91**	-	**99.93**	-	**99.93**	-	**99.92**	-	**99.92**	-	**99.91**	-

mutants, the selected mutation score for S_{base} strategy ranges from 98.23% (on JDepend) to 99.91% (on JodaTime) with the average value of 99.44%. As the sampling ratio increases, all the strategies have higher selected mutation score and lower standard deviation for all subjects. This demonstrate that a user can use the sampling strategies to control the cost-effectiveness of mutation testing: the more mutants sampled, the more precise and stable the results would be.

Second, the studied strategies perform better on larger subjects than on the smaller subjects. For example, when sampling 5% of mutants, S_{meth} achieves the average selected mutation scores ranging from 98.31% to 99.32% for the first four subjects that have fewer than 6000 LOC, while it achieves the average selected mutation scores ranging from 99.69% to 99.92% for all the other seven larger subjects. This demonstrates that using small sampling ratios (e.g., r=5%) of mutants is more beneficial for evaluating test suites for larger subjects. Section 3.3.4 further investigates the effectiveness of sampling mutation for ratios even below 5%.

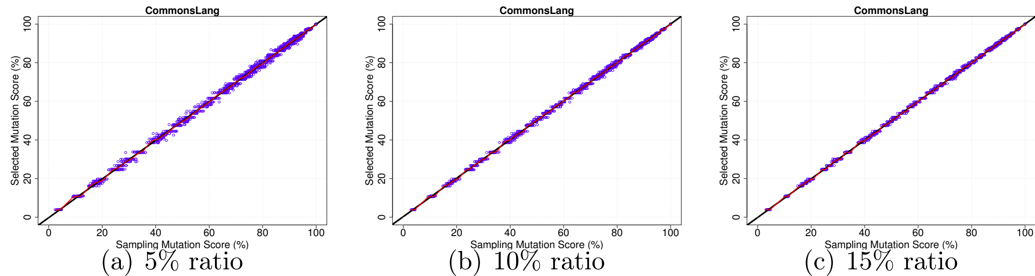
Third, all the strategies perform similarly, but S_{meth} and $S_{meth-mop}$ tend to perform the best of all the strategies for the majority of the subjects. Moreover, the additional use of mutation operator information in $S_{meth-mop}$ does not make it outperform S_{meth}. This demonstrates that sampling mutants across different program elements can be a better choice than sampling mutants globally (S_{base}) or across different mutation operators (S_{mop}). S_{meth} performs better than S_{class} and S_{stmt} likely because sampling at the class level

is too coarse (bringing it closer to S_{base}), while sampling at the statement level is too fine making it select no mutant from some statements (because the number of mutants for each statement is relatively small).

3.3.3.2 Predictive Power for Non-Adequate Test Suites

While the above results showed that *adequate* sampling mutation score implies high selected mutation score, it is uncommon in practice to have adequate test suites. Thus, we further investigate the predictive power of the sampling strategies for *non-adequate* test suites that do not kill all sampled non-equivalent mutants. More precisely, we analyze whether the sampling mutation score is a good predictor of the selected mutation score across a range of test suites, which are almost all non-adequate. Ideally, for all (non-adequate) test suites sampling and selected mutation score would have the same value. In practice, if a test suite achieves selected mutation score MS, the same test suite may achieve sampling mutation score MS' such that $MS < MS'$, $MS = MS'$, or $MS > MS'$. We use three statistical measures to evaluate the predictive power of sampling mutation score for all strategies.

Evaluating Single Test Suite. Originally, mutation testing was proposed as a method for evaluating the quality of test suites by measuring mutation score; the higher mutation score means higher quality. To evaluate a test suite using one of the sampling strategies, we have to ensure that the result obtained on the sampled mutants predicts the result that would be obtained on all the selected mutants. Following previous work [92], we determine how well the independent



(a) 5% ratio (b) 10% ratio (c) 15% ratio

Figure 3.1: Sampling mutation score vs. Selected mutation score, with best fit line (black color) and smoothing spline line (red color), for CommonsLang subject, Meth strategy, and three different sampling ratios

variable (sampling mutation score) predicts the dependent variable (selected mutation score) using a linear regression model. We measure the quality of fit of a model by calculating the adjusted coefficient of determination R^2, which is a statistical measure of how well the regression line approximates the real data points. The value of R^2 is between 0 and 1, where a higher value indicates a better goodness of fit.

We calculate R^2 for each triple (P, S, r) consisting of a subject program, strategy, and ratio. For each sample strategy S, we sample mutants at each ratio r and measure sampling mutation score for the same set of randomly constructed test suites (Section 3.3.2.3). We repeat sampling 20 times to obtain sampling and selected mutation scores for a variety of samples. We then calculate how well the sampling mutation scores from all the 20 sampling runs predict the selected mutation scores by calculating R^2 values[3]. Note that we calculate R^2 for all 20 sampling runs at once (which gives a more robust

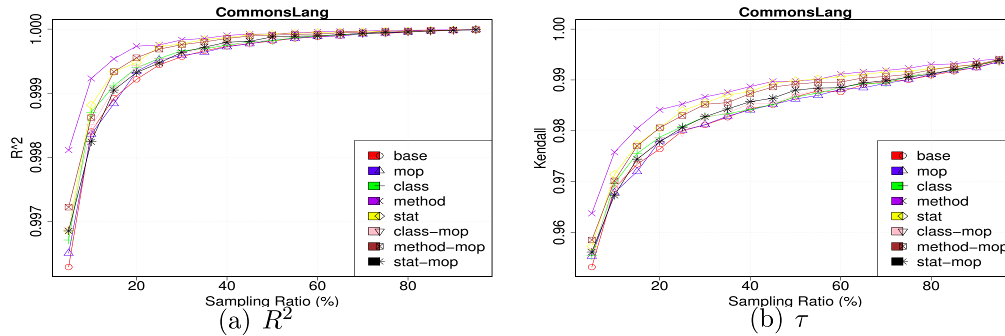
[3]We use R language for statistical computing.

result than calculating R^2 for individual runs and averaging the result over 20 sampling runs). To illustrate, Figure 3.1 shows scatter plots of the sampling mutation score and the selected mutation score for CommonsLang. In each of the three subfigures, the x-axis shows the sampling mutation scores achieved by the test suites on various sampling runs, while the y-axis shows the selected mutation score for the same test suites. There are 20*100=2,000 points on each plot. From the three subfigures, we can see that a higher sampling ratio (r) leads to more stable data points, which can also be seen by smoother splines. However, note that the sampling mutation scores on all sampling runs are close to their selected mutation scores even when $r = 5\%$.

The left part of Table 3.3 shows R^2 values for all strategies with the sampling ratio of 5% on all subjects. (Due to the space limit, the detailed results for the other ratios are not shown but can be found on the project webpage [119].) Column "Subjects" lists the name of the subjects, and columns 2-9 include R^2 values for all 8 sampling strategies. The higher the R^2 value is, the better predictor the sampling strategy is. We find that the R^2 results at the 5% ratio level are already extremely high, e.g., ranging from 0.945 (on JTopas) to 0.998 (on CommonsLang) for the S_{meth} strategy. This further confirms our findings for adequate test suites—the sampling ratio of 5% can be effective for mutation testing in practice. In addition, similar to our findings for adequate test suites, the sampling strategies are less effective for smaller subjects, e.g., the R^2 for the S_{meth} strategy ranges from 0.945 to 0.977 for the first four subjects below 6,000 LOC, while it is over 0.98 for the other

Table 3.3: R^2 and τ correlation values between mutation scores on sampled 5% mutants and on mutants before sampling

Subjects	R^2 correlation								Kendall's τ correlation							
	Base	MOp	Class	Meth	Stmt	Class -MOp	Meth -MOp	Stmt -MOp	Base	MOp	Class	Meth	Stmt	Class -MOp	Meth -MOp	Stmt -MOp
TimeMoney	0.971	0.972	0.975	0.977	0.976	0.975	0.975	0.972	0.874	0.872	0.881	0.888	0.882	0.877	0.880	0.872
JDepend	0.936	0.928	0.927	0.946	0.948	0.933	0.934	0.920	0.784	0.782	0.774	0.791	0.796	0.766	0.789	0.790
JTopas	0.932	0.923	0.909	0.945	0.944	0.943	0.922	0.922	0.845	0.834	0.826	0.856	0.851	0.850	0.836	0.831
Barbecue	0.956	0.951	0.958	0.970	0.964	0.961	0.962	0.946	0.844	0.833	0.849	0.867	0.861	0.854	0.857	0.832
Mime4J	0.991	0.992	0.993	0.994	0.992	0.992	0.993	0.991	0.936	0.937	0.938	0.943	0.936	0.938	0.939	0.933
Jaxen	0.985	0.987	0.983	0.990	0.984	0.986	0.986	0.982	0.894	0.896	0.886	0.910	0.898	0.902	0.901	0.890
XStream	0.993	0.994	0.996	0.996	0.995	0.996	0.996	0.995	0.942	0.946	0.954	0.956	0.949	0.953	0.954	0.948
XmlSecurity	0.982	0.984	0.986	0.986	0.983	0.987	0.984	0.982	0.888	0.892	0.904	0.900	0.894	0.906	0.899	0.896
CommonsLang	0.996	0.996	0.997	0.998	0.997	0.997	0.997	0.997	0.953	0.955	0.955	0.964	0.957	0.956	0.958	0.956
JodaTime	0.996	0.996	0.997	0.998	0.996	0.996	0.997	0.996	0.950	0.950	0.954	0.957	0.952	0.950	0.953	0.949
JMeter	0.982	0.982	0.988	0.989	0.985	0.985	0.988	0.980	0.915	0.917	0.931	0.935	0.925	0.924	0.931	0.913
Avg.	**0.975**	**0.973**	**0.974**	**0.981**	**0.979**	**0.977**	**0.976**	**0.971**	**0.893**	**0.892**	**0.896**	**0.906**	**0.900**	**0.898**	**0.900**	**0.892**



(a) R^2 (b) τ

Figure 3.2: Correlation values for CommonsLang subject, all strategies, and all rates

seven larger subjects. Furthermore, although all the strategies perform well, the S_{meth} strategy slightly outperforms S_{base} and S_{mop} for all the 11 subjects, indicating again that sampling across different program elements can be a better choice than sampling purely randomly from all mutants or sampling across different mutation operators.

To show how the correlation varies when the sampling ratio changes, Figure 3.2(a) shows the R^2 values for all 8 strategies when the sampling ratio increases from 5% to 95% for the subject CommonsLang. The plots for the other subjects look similar and are available on the project webpage [119]. We can draw the following conclusions. First, S_{meth} is slightly better than the other strategies across all sampling ratios, further demonstrating the benefits of sampling mutants across program elements. Second, more importantly, all sampling strategies predict the selected mutation score very well. Across all the programs, strategies, and ratios, the minimum R^2 was 0.909 (for JTopas). Extremely high R^2 gives evidence that sampling mutation is valuable and can

be used for evaluation of test suites. We believe that the results of our study can greatly impact the use of mutation testing in research practice; using sampling mutation testing makes it feasible to evaluate the quality of test suites for large-scale programs.

Comparing Testing Techniques and Test Suites. Mutation testing has also been extensively used in studies that compare testing techniques [19,125, 126]. Commonly, a testing technique or a test suite that has a relatively higher mutation score than another testing technique or test suite is claimed to be better (regardless of the absolute mutation score that it achieves). We thus want to evaluate whether sampling mutation can be used for comparison of testing techniques and test suites, i.e., if a test suite T has a higher *sampling* mutation score than another test suite T', does T have a higher *selected* mutation score than T'? Similar to a previous study [92], we calculate Kendall's τ and Spearman's ρ rank correlation coefficients, which measure the strength of the agreement between two rankings. Both τ and ρ can take values between -1 and 1, where 1 indicates perfect agreement, and -1 indicates perfect disagreement.

To illustrate how τ is computed, consider all the pairs of sampling and selected mutation scores; two pairs (MS_1, MS'_1) and (MS_2, MS'_2) are said to be concordant if $(MS_1 > MS_2 \wedge MS'_1 > MS'_2) \vee (MS_1 < MS_2 \wedge MS'_1 < MS'_2)$ and discordant if $(MS_1 < MS_2 \wedge MS'_1 > MS'_2) \vee (MS_1 > MS_2 \wedge MS'_1 < MS'_2)$; otherwise, the pair is neither concordant nor discordant. Kendall's τ is calculated as the ratio of difference between the number of concordant and

discordant pairs over total number of pairs. In this work we use τ_b, which has a more complex computation because it takes ties into consideration.

We calculate Kendall's τ_b for each triple (P, S, r), following the same procedure as for R^2. Similar with the R^2 measure, we show the τ_b measure for all the strategies on all subjects with the sampling ratio of 5% in the right part of Table 3.3. We also show Kendall's τ_b values for CommonsLang subject, all sampling strategies, and all sampling ratios in Figure 3.2(b). The plots for the other examples look similar and are available on the project webpage [119]. Across all the subjects, all strategies, and all ratios, the minimal value for τ_b in our study was 0.766 (for JDepend).

Considering Table 3.3 and Figure 3.2(b), we can draw similar conclusions as from the R^2 correlation measures. First, all sampling strategies provide very similar result for Kendall's τ. In addition, S_{meth} slightly outperforms S_{base} and S_{mop} for all 11 subjects. Second, all the values are very high, which indicates very strong agreement between rankings. The results for Spearman's ρ show even stronger agreement (details can be found on the project webpage [119]). Based on our study, we believe that the comparison of test suites or testing techniques can be done using sampling mutation.

3.3.3.3 Savings Obtained by Mutation Sampling

Table 3.4 shows the selected mutation testing time for all the mutants generated by Javalanche (recall that Javalanche uses operator-based selection), and the sampling mutation testing time for the sampling ratio of 5% and our

Table 3.4: Selective and sampling mutation testing time

Subject	All Mutants (mmm:ss)	5% Sampled Mutants (mm:ss)			
		Min.	Max.	Avg.	(Pct.)
TimeMoney	7:13	0:54	0:57	0:55	(12.89%)
JDepend	3:02	0:31	0:33	0:32	(17.66%)
JTopas	34:59	1:01	1:12	1:03	(3.02%)
Barbecue	7:35	2:42	2:56	2:46	(36.62%)
Mime4J	181:20	6:44	9:24	8:09	(4.50%)
Jaxen	48:21	2:49	4:03	3:15	(6.75%)
XStream	132:02	4:37	9:49	6:07	(4.64%)
XmlSecurity	53:04	3:17	4:03	3:46	(7.10%)
CommonsLang	74:35	5:12	6:51	6:01	(8.08%)
JodaTime	196:28	12:28	19:03	14:57	(7.61%)
JMeter	57:32	3:42	5:26	4:28	(7.77%)
Avg.	72:22	-	-	4:43	(6.54%)

S_{meth} strategy. Column "Subject" lists the subjects, column "All Mutants" shows the mutant generation and execution times for all the mutants generated by Javalanche, and columns 3-5 list the minimum/maximum/average mutant generation and execution times for the sampling mutation with the sampling ratio of 5% across 20 sampling runs. In column 6 ("Pct."), we also show the ratio of the sampling mutation testing time over the selected mutation testing time. Note that we include the mutant generation time of all selected mutants for both selected mutation and sampling mutation, because our current implementation requires Javalanche to generate all the mutants before sampling. The results show that the sampling mutation testing time, with sampling ratio of 5%, is close to 5% of the selected mutation testing time. We further noticed that the sampling mutation testing time on small subjects tends to be longer than expected 5% of selected mutation time, because for small subjects the tool setup time and the mutant generation time (rather than the mutation execution time) can dominate the total mutation testing time.

However, for the seven larger subjects, the tool setup time and the mutant generation time take insignificant time compared to the total mutation testing time, leading to sampling mutation time from 4.50% to 8.08% of the selected mutation time. On average across all the 11 subjects, the sampling mutation testing time is less than 5 minutes; in contrast, the original Javalanche time is much more and exceeds 70 minutes.

3.3.4 Below 5%

Our experimental results show that it is possible to greatly reduce the number of mutants (e.g., sampling only 5% mutants) while still preserving the mutation score. However, it was not clear whether we can use sampling ratio below 5%. Thus, we additionally collected the experimental results for sampling fewer than 5% mutants. Table 3.5 shows the results for the S_{base} and S_{meth} strategies. The detailed results for all the 8 strategies can be found online [119]. In the table, Column 1 lists all the studied sampling ratios, columns 2-4 list the average selected mutation scores for adequate test suites as well as the average R^2 and the τ correlation values for inadequate test suites by the S_{base} strategy across all subjects. Similarly, columns 5-7 list the corresponding results for the S_{meth} strategy.

The results show that it is possible to have a fairly reliable mutation score even when sampling fewer than 5% mutants. However, by the "rule of 99%" [96], which would require the sampling mutation score to be 99% or higher, the ratio of 3-3.5% is on the borderline for our set of programs and

Table 3.5: Results of sampling below 5% of selected mutants

Ra.	Base			Meth		
	MS.	R^2	τ	MS.	R^2	τ
0.5%	92.24	0.806	0.716	92.02	0.803	0.722
1.0%	96.55	0.896	0.792	96.32	0.905	0.799
1.5%	97.27	0.926	0.819	97.71	0.939	0.837
2.0%	98.21	0.944	0.843	98.48	0.952	0.858
2.5%	98.68	0.955	0.859	98.82	0.964	0.872
3.0%	98.94	0.964	0.874	99.00	0.973	0.885
3.5%	99.07	0.968	0.877	99.22	0.977	0.895
4.0%	99.22	0.974	0.888	99.34	0.978	0.899
4.5%	99.38	0.974	0.893	99.46	0.982	0.907

tests and may not generalize to other programs and tests. In the future, we plan to evaluate whether advanced techniques (e.g., search-based mutant selection [59, 74]) could achieve even smaller sampling ratios. In addition, the results show that S_{meth} outperforms S_{base} in terms of all the three metrics with sampling ratio of greater than 1%, further demonstrating the benefits of our proposed sampling based on program elements.

3.3.5 Threats to Validity

Threats to construct validity. The main threat to construct validity for our study is the set of metrics used to evaluate the mutant sampling strategies. To reduce this threat, we use two widely used metrics, the mutation score metric for adequate test suites [16,96,101,136,149] and the correlation analysis for non-adequate test suites [92]. Our study still inherits a major threat to construct validity: as in those previous studies, we considered all mutants not killed by the original test pool to be equivalent due to the lack of precise techniques for detecting equivalent mutants.

Threats to internal validity. The main threat to internal validity is the potential faults in the implementation of our sampling strategies or in our data analysis. To reduce this threat, the first two authors carefully reviewed all the code for mutant sampling and data analysis during the study.

Threats to external validity. The main threat to external validity is that our results from this study may not generalize to other contexts, including programs, tests, and mutants. To reduce this threat, we select 11 real-world Java programs with various sizes (from 2681 to 36910 lines of code) from various application domains. Note that our study includes more programs than any previous study on selective mutation testing for sequential code [16, 92, 96, 101, 136, 149]. In addition, the 11 programs used in our study are one to two orders of magnitude larger than programs used in similar previous studies.

3.4 Summary

This chapter makes the following contributions:

- **Sampling mutation:** We investigate a simple idea, which we call *sampling mutation*, to reduce the number of mutants generated by operator-based mutant selection: sampling mutation randomly selects from the set of mutants generated by operator-based mutant selection (rather than from the set of mutants generated by all operators [43, 136, 149]); we call the process of obtaining the mutants *sampling*, the percentage of randomly selected mutants the *sampling ratio*, and the resulting set of

mutants a *sample*.

- **Various sampling mutation strategies:** We evaluate 8 sampling strategies; our study is the first to consider random selection based on the program elements (rather than on the mutation operators). Our empirical study shows that although all sampling strategies are effective for mutation testing, sampling based on program elements can provide the most effective results.

- **Extensive study:** We evaluate mutation sampling on 11 real-world Java projects of various sizes (from 2681 to 36910 lines of code) to investigate the effectiveness, predictive power, and savings of sampling mutation. Our study evaluates effectiveness in case of adequate test suites, predictive power in case of non-adequate test suites, and savings in terms of time to generate and execute mutants.

- **Empirical evidence:** The study shows that sampling mutation remains effective and has a high predictive power even while providing high savings. The study shows the cost-effectiveness of applying sampling mutation with various strategies and ratios. Surprisingly, for all our subjects, the experimental results show that sampling only 5% of operator-based selected mutants can still provide a precise mutation score, with almost no loss in precision, while reducing the mutation time to 6.54% on average. Moreover, the study shows that the sampling strategies are more beneficial for larger subjects; as more and more researchers are using

mutants to compare testing techniques, our sampling strategies can help researchers to scale mutation to larger programs by choosing a representative subset of mutants for efficient but effective evaluation.

Chapter 4

Test Selection for Mutation Testing

The previous two chapters (Chapters 2 and 3) introduced two approaches, one each in the areas of regression testing and mutation testing. This chapter introduces our first unification of regression testing with mutation testing: using the test selection technique to incrementally collect mutation testing results. Note that different from regression test selection techniques that only deal with manual changes, our test selection technique needs to deal with two dimensions of changes: both manual and mechanical mutation changes. This chapter is based on our paper presented at the International Symposium on Software Testing and Analysis (ISSTA 2012) [155].

4.1 Background

Despite the potential mutation testing holds for software testing, it primarily remains confined to research settings. One of the main reasons is the costly analysis that underlies the methodology: the requirement to execute many tests against many mutants. A number of techniques aim to scale mutation testing, for example, by selecting a subset of mutants to generate instead of generating all of them [92, 96, 136, 149], by partially executing mutants to

determine whether a test (weakly) kills a mutant [53, 137], and by executing some mutants in parallel [73,85,100]. While these techniques are able to reduce some cost of mutation testing, it still remains one of the most costly software testing methodologies.

Our key insight is that we can amortize this high cost of mutation testing in the context of software systems that undergo evolution by incrementally updating the results for successive applications of mutation testing. Real software systems undergo a number of revisions to implement bug fixes, add new features, or refactor existing code. An application of existing mutation testing techniques to an evolving system would require repeated, independent applications of the technique to each software version, inducing expensive costs for every version. Our approach utilizes the mutation testing results on a previous version to speed up the mutation testing for a subsequent version. Our approach opens a new direction for reducing the cost of mutation testing; it is orthogonal to the previous techniques for optimizing mutation testing, and it is applicable together with these previous techniques.

This chapter presents *Regression Mutation Testing (ReMT)*, a novel technique that embodies our insight. ReMT identifies mutant-test pairs whose execution results (i.e., whether the test killed the mutant or not) on the current software version can be reused from the previous version without re-executing the test on the mutant. ReMT builds on the ideas from regression test selection techniques that traverse control flow graphs of two program versions to identify the set of *dangerous edges* which may lead to different test behaviors in the

new program version [51,102,114]. More precisely, ReMT reuses a mutant-test result if (1) the execution of the test *does not* cover a dangerous edge *before* it reaches the mutated statement for the first time and (2) the execution of the test *cannot* reach a dangerous edge *after* executing the mutated statement. ReMT determines (1) with dynamic coverage and determines (2) with a novel static analysis for *dangerous-edge reachability* based on Context-Free-Language (CFL) reachability.

As an additional optimization to our core ReMT technique, we introduce *Mutation-specific Test Prioritization* (MTP). For each mutant, MTP reorders the tests that need to be executed based on their effectiveness in killing that mutant on previous versions and their coverage of the mutated statement. Combining ReMT with MTP can further reduce the time to kill the mutants.

4.2 Definitions

This section describes some core concepts in mutation testing (Section 4.2.1) and regression testing (Section 4.2.2) that are used in this chapter. It also provides some basic definitions that we use to present our Regression Mutation Testing (Section 4.2.3).

4.2.1 Mutation Testing

Mutation testing, first proposed by DeMillo et al. [32] and Hamlet [46], is a fault-based testing methodology that is effective for evaluating and improving the quality of test suites. Given a program under test, P, mutation

testing uses a set of *mutation operators* to generate a set of *mutants* M for P. Each mutation operator defines a rule to transform program statements, and each mutant $m \in M$ is the same as P except for a statement that is transformed. Given a test suite T, a mutant m is said to be *killed* by a test $t \in T$ if and only if the execution of t on m produces a different result from the execution of t on P. Conceptually, mutation testing builds a mutant execution matrix:

Definition 4.2.1. *A* ***mutant execution matrix*** *is a function* $M \times T \rightarrow \{\texttt{U}, \texttt{E}, \texttt{N}, \texttt{K}\}$ *that maps a mutant* $m \in M$ *and a test* $t \in T$ *to: (1)* `U` *if* t *has not been executed on* m *and thus the result is unknown, (2)* `E` *if the execution of* t *cannot reach the mutated statement in* m *(and thus* m *cannot be killed by test* t*), (3)* `N` *if* t *executes the mutated statement but does not kill* m*, and (4)* `K` *if* t *kills* m*.*

The aim of our ReMT technique is to speed up the computation of the mutant execution matrix for a new program version based on the mutant execution matrix for an old program version. Note that for the very first version the old matrix has all cells as `U` because there is no previous version. For future versions, the old matrix may in the limit be *full*, having no cell as `U`. However, our ReMT technique does *not* require such full matrices. Indeed, to compute the mutation score for a given program, for each mutant m, it suffices that the matrix has (1) *at least one* cell as `K` (while others can be `E`, `N`, or even `U`), or (2) *all* cells as `E` or `N` (indicating that the test suite T does not kill m).

Some existing mutation testing tools, such as Javalanche [122] and Proteum [29], support two mutation testing scenarios: (1) *partial mutation testing* – where a mutant is only run until it is killed and thus the matrix may have some `U` cells; and (2) *full mutation testing* – where a mutant is run against each test and thus the mutant execution matrix has no `U` cells. Our ReMT technique is applicable for both scenarios.

4.2.2 Regression Testing

A key problem studied in regression testing is Regression Test Selection (RTS): determine how changes between program versions influence regression tests and select to run only tests that are related to changes. RTS techniques [51, 102, 114] commonly use the control-flow graph (CFG) and its extended forms, e.g., the Java Interclass Graph [51], to represent program versions and analyze them. A typical RTS technique first traverses CFGs of two program versions using depth-first search (DFS) to identify the set of *dangerous edges*, E_Δ, i.e., the edges which may cause the program behavior to change in the new program version. Then, for each test t in the regression test suite, the technique matches its coverage information on the old version with the set of dangerous edges E_Δ to determine whether t could be *influenced* by the dangerous edges.

Following previous work [51, 102], we consider RTS techniques that use inter-procedural CFGs:

Definition 4.2.2. *An* ***inter-procedural CFG*** *of a program is a directed*

graph, $\langle N, E\rangle$, where N is the set of CFG nodes, and $E : N \times N$ is the set of CFG edges.

Each inter-procedural CFG has several intra-procedural CFGs:

Definition 4.2.3. *An **intra-procedural CFG** within an inter-procedural CFG $\langle N, E\rangle$ is a subgraph $\langle N_i, E_i\rangle$, where $N_i \subseteq N$ and $E_i \subseteq E$ denote edges that start from nodes in N_i. Each intra-procedural CFG has a unique entry node and a unique exit node.*

Note that E_i includes edges that are method invocation edges connecting invocation nodes in N_i with entry nodes of other intra-procedural CFGs, as well as edges that are return edges connecting the exit node with return nodes of other intra-procedural CFGs. Thus, $E_i \subseteq N_i \times N$. Moreover, each invocation node can be linked to different target methods based on the possible receiver object types, and thus each invocation edge is labeled with a run-time receiver object type to identify dangerous edges caused by dynamic dispatch changes.

Traditional RTS techniques [51, 102, 114] explore CFG nodes of two programs versions using DFS search to determine the equivalence of node pairs by examining the syntactic equivalence of the associated statements. They determine the set of dangerous edges:

Definition 4.2.4. *The set of **dangerous edges** between two inter-procedural CFGs $\langle N, E\rangle$ and $\langle N', E'\rangle$ is the set of edges $E_\Delta \subseteq E$ whose target nodes have been changed to non-equivalent nodes or whose edge labels have been changed.*

4.2.3 Regression Mutation Testing

To reuse mutation testing results from an old program version for the new program version, ReMT maintains a mapping between the mutants of the two program versions. This mutant mapping is based on the CFG node mapping:

Definition 4.2.5. *For two inter-procedural CFGs $\langle N, E\rangle$ and $\langle N', E'\rangle$, the* ***CFG node mapping*** *is defined as function* `mapN`*: $N' \rightarrow N \cup \{\perp\}$ that maps each node in N' to its equivalent node in N or to $\perp$ if there is no such equivalent node.*

Note that the node mapping is constructed during the DFS search by RTS for identifying dangerous edges.

The mapping between mutants of two program versions is defined as follows:

Definition 4.2.6. *For two program versions P and P' and their corresponding sets of mutants M and M',* ***mutant mapping*** *between P and P' is defined as function* `mapM`*: $M' \rightarrow M \cup \{\perp\}$, that returns mutant $m \in M$ of P for mutant $m' \in M'$ of P', if (1) the mutated CFG node $n_{m'}$ of m' maps to the mutated CFG node n_m of m (i.e., $n_m = \texttt{mapN}(n_{m'})$) and (2) m' and m are mutated by the same mutation operator at the same location; otherwise,* `mapM` *returns $\perp$.*

The traditional RTS techniques [51, 114] compute influenced tests by intersecting edges executed by the tests on the old program version with the

dangerous edges. However, such computation of intersection for original, unmutated programs does *not* work for regression *mutation* testing, because the test execution path for each mutant may differ from the path for the original program. Therefore, for ReMT, we introduce a static analysis for checking the reachability of dangerous edges for each mutant when it is executed by each test. Our ReMT technique computes the set of dangerous edges reachable from each node n along the execution of each test t in the test suite T based on inter-procedural CFG traversal:

Definition 4.2.7. *For an inter-procedural CFG $\langle N, E \rangle$ with a set of dangerous edges E_Δ, the **dangerous-edge reachability** for node $n \in N$ with respect to test $t \in T$ is a predicate* `reach` $\subseteq N \times T$*;* `reach`(n, t) *holds iff an execution path of t could potentially go through node n and reach a dangerous edge after n.*

Note that a node n can have different reachability results with respect to different tests, i.e., `reach`(n, t) for a test t may differ from `reach`(n, t') for another test t'.

Our ReMT technique also utilizes the test coverage of CFG nodes and edges. Specifically, we utilize *partial* test coverage on CFG nodes and edges before a given CFG node is executed:

Definition 4.2.8. *For a program with CFG $\langle N, E \rangle$, **test coverage** is a function* `trace`*: $T \times (N \cup \{\bot\}) \rightarrow 2^{N \cup E}$ that returns a set of CFG nodes $N_{sub} \subseteq N$*

and a set of CFG edges $E_{sub} \subseteq E$ covered by test t before the first execution of node $n \in N$; $\texttt{trace}(t, \perp)$ *is the set of all nodes and edges covered by test t.*

Note that this notation allows simply using $\texttt{trace}(t, \texttt{mapN}(n_m))$ to evaluate to (1) the set of nodes and edges covered before n_m if there is a corresponding mapped node for n_m, and (2) the set of all nodes and edges covered by t if there is no mapped node.

4.3 Example

Figure 4.1 shows two versions of a small program, `Account`, which provides basic bank account functionality. Lines 20 and 25 in the old version are changed into lines 21 and 26 in the new version, respectively. As the change on line 25 would cause the regression test suite (`TestSuite`) to fail on `test3`, the developer also modifies `test3` to make the suite pass.

Figure 4.2 shows the inter-procedural CFG. We depict the changed nodes in gray; *dangerous edges* are the edges incident to the gray nodes (e.g., $\langle 19, 20\rangle$, $\langle 11, 25\rangle$, and $\langle \texttt{return}, 40\rangle$). The CFG consists of six intra-procedural sub-CFGs, which are connected using inter-procedural invocation and return edges. Each invocation site is represented by an invocation node and a return node, which are connected by a virtual path edge.

To illustrate ReMT, consider the mutants in Table 4.1. Assuming we already have some mutant execution results from the old version, we collect mutant execution results for the new version incrementally. We demonstrate

```
 1 public class Account {
 2   double balance;  double credit;
 3   public Account(double b,double c){
 4     this.balance=b; // deposit balance
 5     this.credit=c; // consumed credit
 6   }
 7   public double getBalance(){
 8     return balance;
 9   }
10   public String withdraw(double value){
11     if(value>0){
12       if(balance>value){//deposit enough?
13         balance=balance-value;
14         return "Success code: 1";
15       }
16       double diff=value-balance;
17       if(credit+diff<=1000){//credit enough?
18         balance=0;
19         credit=credit+diff;
20       - return "Success code:  1";
21       + return "Success code:  2";
22       }
23       else return "Error code: 1";
24     }
25   - return "Error code:  1";
26   + return "Error code:  2";
27   }
28 }
29 public class TestSuite {
30   public void test1(){
31     Account a=new Account(20.0,0.0);
32     assertEquals(20.0,a.getBalance());}
33   public void test2(){
34     Account a=new Account(20.0,0.0);
35     String result=a.withdraw(10.0);
36     assertEquals("Success code: 1",result);}
37   public void test3(){
38     Account a=new Account(20.0,0.0);
39     String result=a.withdraw(10.0);
40    - assertEquals("Error code:  1",result);
41   + assertEquals("Error code:  2",result);}
42 }
```

Figure 4.1: Example code evolution and test suite.

both full mutation testing and partial mutation testing scenarios. Following Definition 4.2.1, the example input matrices of the old version in both scenarios are shown in the top parts of tables 4.2 and 4.3. In both scenarios, we initialize the mutation results of the new program version as a new mutant execution matrix with all `U` elements to denote that the mutant execution results are

Table 4.1: Mutants for illustration.

Mutant	Mutated Location	Mutant Statement
m_1	n_4	`this.balance=0`
m_2	n_5	`this.credit=0`
m_3	n_8	`return 1`
m_4	n_{11}	`if(value<=0)`
m_5	n_{12}	`if(balance<value)`
m_6	n_{13}	`balance=balance+value`
m_7	n_{13}	`balance=balance*value`
m_8	n_{13}	`balance=balance/value`
m_9	n_{14}	`return ""`

Table 4.2: Incrementally collecting full matrix.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
t1	K	N	K	N	N	N	N	N	N
t2	N	N	N	K	N	N	N	N	K
t3	N	N	N	K	N	N	N	N	N
t1	K	N	K	E	E	E	E	E	E
t2	U	U	E	U	U	N	N	N	K
t3	U	U	E	U	E	E	E	E	E

Table 4.3: Incrementally collecting partial matrix.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
t1	K	N	K	N	N	N	N	N	N
t2	U	N	U	K	N	N	N	N	K
t3	U	N	U	U	N	N	N	N	U
t1	K	N	K	E	E	E	E	E	E
t2	(U)	U	E	U	U	N	N	N	K
t3	(U)	U	E	U	E	E	E	E	E

initially unknown. In total, at most 27 mutant-test executions are needed for computing each mutant execution matrix for the new version.

To reduce the number of mutant-test executions, several mutation testing tools [6,70,122] utilize the following fact: when a test executed on the original, unmutated program does not cover the mutated statement of a mutant, then that test cannot kill that mutant. One can thus filter out a set of tests for each mutant (or dually a set of mutants for each test). Figure 4.2 highlights the execution traces of `test1`, `test2`, and `test3` after evolution with bold solid (red) lines, bold dashed (blue) lines, and bold dotted (gray) lines, respectively. Here, for example, any mutant that does not occur on the nodes

in bold solid (red) lines cannot be killed by `test1`, and any mutant that does not occur on the nodes in bold dashed (blue) lines cannot be killed by `test2`. The matrices for the new version can then be updated with `E` elements to denote a mutant that cannot be killed by a test because its mutated statement is not reached by the test. The light-gray cells in tables 4.2 and 4.3 show the cells updated with `E`'s. Now, 14 mutant-test executions (i.e., cells not marked as `E`'s) are required for computing the full matrix, and at most 14 executions are required for computing the partial matrix.

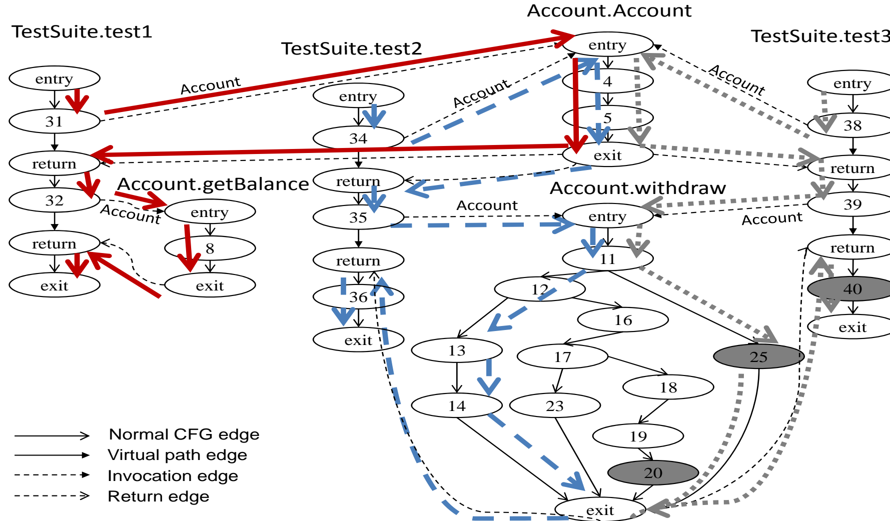


Figure 4.2: Inter-procedural CFG for the example.

To further reduce the number of mutant-test executions, our ReMT leverages program evolution information. For instance, mutants m_6, m_7, m_8, and m_9 would need to be executed against `test2` when not consider evolution; however, we can compute that those mutants cannot modify the `test2`'s execution trace to reach any dangerous edge, because there is no CFG path from

the nodes where those mutants occur (i.e., n_{13} and n_{14}) to the evolved code (i.e., n_{20}, n_{25}, and n_{40}). Therefore, the mutation testing results for mutants on these two nodes cannot differ from their previous results for the program before evolution, and these results can be directly reused from the old mutant execution matrix.

We use a static analysis to determine which dangerous edges can be reached by mutants. It is important to point out that this analysis is done *with respect to each test* (Definition 4.2.7) because the results can differ for different tests. For example, consider node n_4 and mutant m_1. Although n_4 *cannot* reach any dangerous edge through inter-procedural CFG traversal with respect to `test1`, n_4 *can* potentially reach dangerous edges through traversal from `test2`. When executing m_1 on `test2`, the execution path takes a different branch at n_{12} than the execution path takes when executing the unmutated new version. Thus m_1 executes the dangerous edge $\langle 19, 20 \rangle$, which actually causes m_1 to be killed for the new version although it is not killed for the old version. Therefore, our dangerous-edge reachability analysis considers potential execution paths for each test. The matrices for the new version can now be updated with history information from the old version (shown as dark-gray cells in tables 4.2 and 4.3). Thus, only 7 mutant-test executions (i.e., cells with `U`'s) are required for obtaining the full matrix, and at most 5 executions are required for obtaining the partial matrix (the two `U`s within brackets of m_1 do not need to be filled because m_1 is already killed by `test1`). In sum, for this example, compared with the state-of-the-art mutation testing techniques,

ReMT reduces the number of mutant-test executions 2X (7 vs. 14) for full mutation testing and over 2X for partial mutation testing (depending on the order in which tests are executed for a mutant as we discuss later).

4.4 Approach

4.4.1 Overview

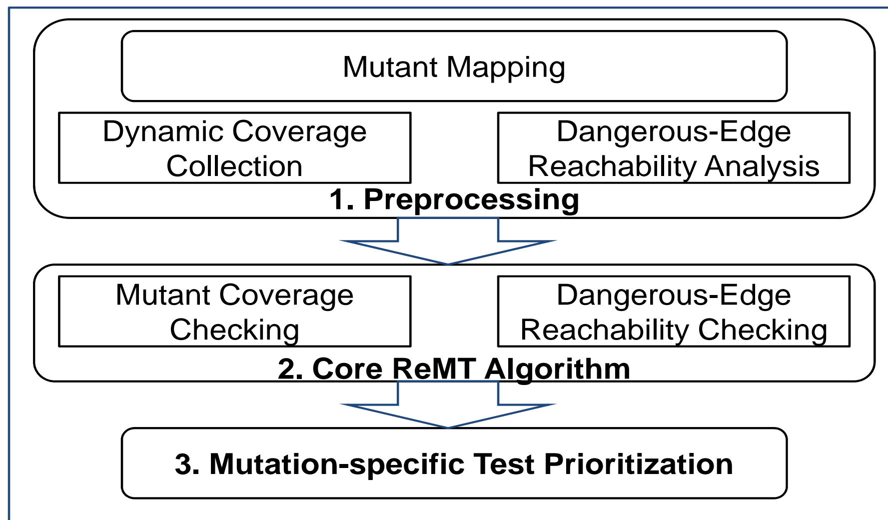


Figure 4.3: General approach of ReMT.

This section presents regression mutation testing (ReMT). Figure 4.3 shows the three key components. The Preprocessing component (Section 4.4.2) builds a mapping between the mutants of the two versions and gathers initial data for the checking performed by the core ReMT component (Section 4.4.3), which consists of two steps: *mutant-coverage checking* and *dangerous-edge reachability checking*. Mutant-coverage checking follows previous work [6, 70, 122] in using the coverage information of all tests on the new program ver-

sion to select the subset of tests that actually execute the mutated statement and thus may kill the mutant for the new version. For the selected tests that do not have execution history (i.e., are newly added tests), ReMT executes them for gathering mutation testing results for the mutant. For the selected tests that have execution history, ReMT's dangerous-edge reachability checking determines whether the mutation results can be reused. More precisely, a mutant-test result can be reused if (1) no dangerous edge is executed from the beginning of the test to the mutated statement and (2) no dangerous edge can be executed from the mutated statement to the end of the test. For (1), ReMT uses dynamic coverage, and for (2), ReMT uses a novel dangerous-edge reachability analysis. When possible, ReMT directly reuses execution results from their previous execution on the mapping mutant of the old version. Finally, as the order of test execution matters for killing mutants faster, ReMT's Mutation-specific Test Prioritization component (Section 4.4.4) reorders tests to further optimize regression mutation testing.

4.4.2 Preprocessing

Preprocessing consists of mutant mapping, coverage collection, and dangerous-edge reachability analysis. Coverage collection uses the common code instrumentation, so we present details of the mutant mapping and dangerous-edge reachability analysis. The construction of mutant mapping also identifies dangerous edges that are used in dangerous-edge reachability analysis.

4.4.2.1 Mutant Mapping

Following existing regression test selection (RTS) techniques [51, 102, 114], ReMT uses control-flow graph (CFG) to represent program versions and identifies program changes as dangerous edges. ReMT uses a standard depth-first search (DFS) for detecting dangerous edges [51,114]. In addition, ReMT's CFG comparison algorithm builds `mapN`, which stores the CFG node mapping between the two program versions (Definition 4.2.5) and is used to calculate mutant mapping `mapM` (Definition 4.2.6). When visiting a node pair, the CFG comparison algorithm first marks the node pair as visited and puts the matched node pair into `mapN`. Then, the algorithm iterates over all the outgoing edges of the node pair: (1) for the edges without matched labels or target nodes, the algorithm puts the edges into the dangerous edge set E_Δ and backtracks the traversal along those edges; (2) for the matched edges (i.e., when both labels and target nodes are matched) whose target nodes have been visited, the algorithm backtracks; (3) for the matched edges whose target nodes have not been visited, the algorithm recursively traverses the target node pairs. Finally, the algorithm returns all dangerous edges E_Δ, node mapping `mapN`, and mutant mapping `mapM` between the old and new program versions.

4.4.2.2 Dangerous-Edge Reachability Analysis

Given a test t, a node n in the CFG, and a set of dangerous edges E_Δ, dangerous-edge reachability computes if n can reach a dangerous edge with respect to t (Definition 4.2.7). We reduce the dangerous-edge reachability

problem to the *Context-Free-Language (CFL) Reachability* [89, 112] problem. The use of CFL-reachabillity on the *inter*-procedural CFG allows us to obtain more precise results than we would obtain by running a simple reachability that would mix invocation and return edges. (For example, in Figure 4.1, a naïve reachability could mix the invocation of the `Account` constructor from `test1` with the return from the `Account` constructor to `test3` and could then (imprecisely) find that `test1` can reach a dangerous edge that ends in n_{40}.) In CFL-reachability, a path is considered to connect two nodes only if the concatenation of the labels on the edges of the path is a word in a particular context-free language:

Definition 4.4.1. *Let L be a context-free language over alphabet Σ and G be a graph whose edges are labeled with elements of Σ. Each path in G defines a word over Σ formed by concatenating the labels of the edges on the path. A path in G is an* L-path *if its word is a member of the language L.*

We reduce our dangerous-edge reachability analysis to a CFL-reachability problem as follows. For a CFG $\langle N, E\rangle$ with I invocation sites, the alphabet Σ has symbols $(_i$ and $)_i$ for each i from 1 to I, as well as two unique symbols, e and d. Following the existing inter-procedural program analysis [89, 112], our analysis labels all the *intra*-procedural edges with e, and for each invocation site i labels its invocation edge and return edge with $(_i$ and $)_i$, respectively. In contrast with the existing techniques, our analysis further labels all dangerous edges with d. A path in $\langle N, E\rangle$ is a *matched path* iff the path's word is

in the language $L(matched)$ of balanced-parenthesis strings according to the following context-free grammar:

$$\begin{aligned} matched &\rightarrow matched \;\; matched \\ &\mid (_i \; matched \;)_i \qquad \forall i : 1 \leq i \leq I \\ &\mid e|d|\varepsilon \end{aligned} \tag{4.1}$$

The language $L(dangerous)$ that accepts all possible valid execution paths to dangerous edges is defined as:

$$\begin{aligned} dangerous &\rightarrow matched \;\; dangerous \\ &\mid (_i \; dangerous \qquad \forall i : 1 \leq i \leq I \\ &\mid d \end{aligned} \tag{4.2}$$

The language $L(dangerous)$ is a language of partially balanced parentheses, which allows representing that the execution might go into some deeper stacks and not return as long as it encounters a dangerous edge. A path is a *dangerous path* iff the path's word is in the language $L(dangerous)$.

The problem of determining all possible nodes that can reach dangerous edges with respect to each test is transformed into the problem of finding all the possible nodes reachable from the root node of each test in the language $L(dangerous)$. For a node n and a test t, $\texttt{reach}(n, t)$ holds if n is reachable from the root node of t in the language $L(dangerous)$; otherwise $\texttt{reach}(n, t)$ does not hold (Definition 4.2.7). Our implementation uses the general dynamic-programming algorithm to efficiently solve the CFL-reachability

problem [89] and record all the nodes that can appear on the dangerous paths for each test. Note that the analysis for one test gives the dangerous-edge reachability for *all* mutants that the test can execute, i.e., ReMT does not repeat this static analysis for each mutant-test pair. Also note that we apply dangerous-edge reachability analysis on the *old* (not new) program version.

4.4.3 ReMT Algorithm

Algorithm 2 shows our core ReMT algorithm, which supports both the partial mutation testing and full mutation testing scenarios. The underlined statements are specific to the partial mutation testing scenario. Note that ReMT does not require a full input matrix on old version. The algorithm expects that preprocessing (Section 4.4.2) has been performed, which enables the use of mutant mapping `mapM` in line 12, mutant-coverage checking (denoted as `MCoverageCheck`) in line 5, and dangerous-edge reachability checking (denoted as `DReachabilityCheck`) in line 10.

4.4.3.1 Basic Algorithm

Lines 2-19 iterate over the mutants of P' and the tests in T' to get the mutation testing results. For each mutant m, lines 3 and 4 first initialize all the test results as `U`. Line 5 applies mutant-coverage checking [6, 70, 122] between P' and m to select the subset of tests within test suite T' that cover the mutated node n_m of m on P'. Formally, the mutant-coverage checking is

Algorithm 2: Algorithm for ReMT

Input: P and P', old and current program versions; M and M', the mutants for P and P'; T and T', test suites for P and P'; `matrix`, the mutant execution results for P.

Output: `matrix'`, the mutant execution results for P'.

Require: Preprocessing (Mutant Mapping, Coverage Collection, and Dangerous-Edge Reachability Results).

1 **begin** ReMT
2   **foreach** *mutant* m : M′ **do**
3   **foreach** *test* t : T′ **do**
4       $\texttt{matrix}'(m, t) \leftarrow \texttt{U}$ // initialization
5     $T_c \leftarrow \texttt{MCoverageCheck}(T', P', m)$
6   *killed* ← `false`
7   **foreach** *test* t : $\mathrm{T}' - \mathrm{T_c}$ **do**
8       $\texttt{matrix}'(m, t) \leftarrow \texttt{E}$ // t cannot kill the mutant
9     $T_c' \leftarrow T_c \cap T$ // tests with execution history
10     $T_r \leftarrow \texttt{DReachabilityCheck}(E_\Delta, T_c', m, P, P')$
11   **foreach** *test* t : $\mathrm{T_c'} - \mathrm{T_r}$ **do**
12       $\texttt{matrix}'(m, t) \leftarrow \texttt{matrix}(\texttt{mapM}(m), t)$
13   **if** $\texttt{matrix}'(\mathrm{m}, \mathrm{t}) = \texttt{K}$ **then**
14   *killed* ← `true` // m has been killed
15   **if** killed =`true` **then** **continue**
16   **foreach** *test* t : $\mathrm{T_c}$ **do**
17   **if** $\texttt{matrix}'(\mathrm{m}, \mathrm{t}) = \texttt{U}$ **then**
18         $\texttt{matrix}'(m, t) \leftarrow \texttt{Execution}(t, m)$
19   **if** $\texttt{matrix}'(\mathrm{m}, \mathrm{t}) = \texttt{K}$ **then** **continue**
20   **return** `matrix'`// return mutation testing result

computed as:

$$\texttt{MCoverageCheck}(T', P', m) = \{t \in T' | n_m \in \texttt{trace}'(t, \perp)\}$$

where $\texttt{trace}'(t, \perp)$ is the entire coverage of t on P' (Definition 4.2.8). The tests that do not cover n_m in P' cannot kill m, so lines 7-8 assign `E` to all such tests. Line 9 stores in T_c' the tests in T_c that have execution history (i.e., the tests that also exist in the old suite of P). Line 10 finds the tests from T_c' that can potentially reach dangerous edges in E_Δ when executing the mutated statement n_m (Section 4.4.3.2). For the tests in $T_c' - T_r$ that cannot reach any dangerous edge, ReMT directly copies the execution results from

the corresponding mapping mutant of P to the execution results on m of P' (lines 11-14). Note that when the input matrix is partial, ReMT may also copy `U` values to the new matrix. When ReMT is applied in the partial mutation testing scenario, it sets the flag *killed* to `true` if the mapping mutant has been killed for P and proceeds to the next mutant (line 15). Lines 16-19 run all the tests in T_c with value `U` on m (i.e., the newly added tests without execution history in T_c, the potentially influenced tests that could reach a dangerous edge, and the tests whose results are copied as `U`s from the input matrix). When ReMT is applied in the partial mutation testing scenario, it terminates the test execution for m as soon as m is killed by some test. Finally, line 20 returns the mutation testing results for P'.

4.4.3.2 Dangerous-Edge Reachability Checking

Algorithm 2 invokes `DReachabilityCheck` at line 10 to perform dangerous-edge reachability checking. After computing T_c', the set of tests that execute the mutated statement for m and have execution history, ReMT further computes T_r, the tests from T_c' that can potentially reach dangerous edges E_Δ between P and P'. There are two types of tests from T_c' that can potentially reach E_Δ: (1) the tests that directly execute edges in E_Δ before the first execution of the mutated CFG node n_m; and (2) the tests that can potentially reach edges in E_Δ from the mutated CFG node. The first type of tests is easily identified by intersecting E_Δ with edge coverage before the mutated node, while the second type of tests is identified by checking the reachability

to dangerous edges from the mutated node with respect to the corresponding test. Formally, we define the reachability checking as follows:

$$\texttt{DReachabilityCheck}(E_\Delta, T'_c, m, P, P') =$$
$$\{t \in T'_c | \texttt{trace}(t, \texttt{mapN}(n_m)) \cap E_\Delta \neq \emptyset \ \vee \texttt{reach}(\texttt{mapN}(n_m), t)\}$$

where `trace` denotes the test coverage for P, and `reach` denotes the reachability for dangerous edges. Note that the checking is performed on the old version P because E_Δ are edges from P. Thus, we need to map n_m back to its mapped node in P. (If there is no mapped node, there must be a dangerous edge before n_m and thus $\texttt{trace}(t, \texttt{mapN}(n_m)) = \texttt{trace}(t, \perp)$ is overlapped with E_Δ.)

4.4.4 Mutation-Specific Test Prioritization

We next present *mutation-specific test prioritization* (MTP) that aims to prioritize remaining tests for each mutant to kill it as early as possible in the partial mutation testing scenario. Given a mutant m of a program P' (that evolved from P), MTP calculates the priority of each test based on its coverage of the mutated statement as well as the mutation testing history. Formally, the priority of test t for m is calculated as:

$$\texttt{Pr}(t, m) = \begin{cases} \langle 1, \texttt{CovNum}(t, n_m) \rangle, & \text{if } \texttt{matrix}(\texttt{mapM}(m), t) = \texttt{K} \\ \langle 0, \texttt{CovNum}(t, n_m) \rangle, & \text{otherwise.} \end{cases}$$

The priority is a pair whose first element represents the mutation testing result for the test on the corresponding mutant of the old version P (`1` if killed,

`0` otherwise), and the second element, `CovNum`(t, n_m), is the number of times the test covers the statement (in the unmutated new version P') to be mutated (to form the mutant). Note that if the test does not have an execution history on the old version (e.g., the test is newly added or was not executed in the partial scenario), or the mutant does not have a mapping mutant for the old version, the first element is set to `0`.

For each mutant, ReMT prioritizes tests lexicographically based on their priority pairs. The tests with first elements set to 1 are executed earlier based on the intuition that a test that kills a mutant in the old version might also kill its mapping mutant in the new version. The tests with second elements that indicate more execution are executed earlier based on the intuition that a mutant is more likely to be killed if its mutated statement was covered more times by a test. If two tests have the same priority values, ReMT executes them according to their order in the original test suite.

4.4.5 Discussion and Correctness

While we presented ReMT for Java and JUnit tests, it is also applicable for other languages and test paradigms. When the test code does not have unit tests, our dangerous-edge reachability analysis can be directly applied on the `main` method of the system under test. Note that ReMT only works for traditional mutation operators that change statements in methods. In the future, we plan to support class-level mutation operators [82] that can change class hierarchy.

We need to show that for each mutant-test result reused from the old version, the same result would be obtained if the corresponding mutant and test were run on the new version. Intuitively, ReMT is correct as it works similarly to regression test selection: for each mutant, any test that might potentially reach dangerous edges is selected.

Theorem 4.4.1. *For every mutant m of P', the influenced test set T_r that ReMT (Line 10) selects from T'_c is such that every test t not selected (i.e., $t \in T'_c - T_r$) has an equivalent execution on the corresponding mutant* `mapM`(m) *of P.*

Proof. By contradiction. Assume some test t is not selected in T_r for some m, but t has no equivalent execution on mutants of P. There are two cases to consider: (1) mutant m does not have a corresponding mutant on P, i.e., `mapM`$(m) = \perp$; or (2) mutant m has a corresponding mutant `mapM`(m) on P, but t can potentially diverge into different executions on m and `mapM`(m).

Case I: According to the definition of mutant mapping (Definition 4.2.6), if m does not have a corresponding mutant on P, then the mutated node n_m for m does not have a mapping node on P, i.e., `mapN`$(n_m) =\perp$. According to the DFS-based node-mapping construction (Section 4.4.2.1), there is no corresponding node for n_m in P or no CFG path that leads to the corresponding node of n_m without executing a dangerous edge. Let the test execution trace for t on P be $n1\langle n_1, n_2\rangle n_2\langle n_2, n_3\rangle, \ldots, \langle n_{l-1}, n_l\rangle n_l$. If this trace covered no dangerous edge, then t would have exactly the same execution trace on P and P', and thus

could not cover n_m, which would be inconsistent with t being selected in T'_c. Therefore, some covered edge $\langle n_{i-1}, n_i \rangle$ $(2 \leq i \leq l)$ must be a dangerous edge, i.e., $\langle n_{i-1}, n_i \rangle \in E_\Delta$. Thus, $\texttt{trace}(t, \texttt{mapN}(n_m)) \cap E_\Delta = \texttt{trace}(t, \perp) \cap E_\Delta \neq \emptyset$, and based on (the first disjunct of) `DReachabilityCheck` (Section 4.4.3.2), t would be selected in T_r. Contradiction.

Case II: According to the definition of mutant mapping (Definition 4.2.6), if m has a corresponding mutant on P, then the mutated node n_m for m has a mapping node on P, i.e., $\texttt{mapN}(n_m) \neq \perp$. Hence, we can represent the test execution trace for t on P as $n_1\langle n_1, n_2\rangle n_2 \ \ldots \ \langle n_i, \texttt{mapN}(n_m)\rangle \texttt{mapN}(n_m)$ $\ldots \ \langle n_{l-1}, n_l\rangle n_l$. Since we assume that t leads to different executions on m and its corresponding mutant $\texttt{mapM}(m)$, t should cover some changed CFG nodes or edges on m; otherwise, the execution of t would not differ between m and $\texttt{mapM}(m)$. There are three sub-cases: (i) t executes changed CFG nodes or edges *during* the first execution of the mutated node n_m; (ii) t executes changed CFG nodes or edges *before* the first execution of the mutated node n_m; or (iii) t executes changed CFG nodes or edges *after* the first execution of the mutated node n_m.

Sub-case (i): If t executes changed CFG nodes or edges during the first execution of n_m, then n_m itself is changed during evolution. Based on Definition 4.2.5, there would not be a corresponding CFG node for n_m on P. Therefore, there also would not be a corresponding CFG mutant for m on P (based on Definition 4.2.6). Contradiction.

Sub-case (ii): If t executes changed CFG nodes or edges before the

first execution of n_m, then there exists an edge $\langle n_{j-1}, n_j \rangle$ $(2 \leq j \leq i)$ before the execution of $\texttt{mapN}(n_m)$ in the trace of t on P such that $\langle n_{j-1}, n_j \rangle \in E_\Delta$; otherwise, the execution of t before n_m would be exactly the same as its execution on $\texttt{mapM}(m)$ and could not cover any changed nodes or edges before n_m. Therefore, $\texttt{trace}(t, \texttt{mapN}(n_m)) \cap E_\Delta \neq \emptyset$, and based on (the first disjunct of) `DReachabilityCheck` (Section 4.4.3.2), t would be selected in T_r. Contradiction.

Sub-case (iii): If t executes changed CFG nodes or edges after the first execution of n_m, then there exists a dangerous edge $e \in E_\Delta$ such that $e \in \texttt{reach}(\texttt{mapN}(n_m), t)$; otherwise, the execution of t on m after n_m would be exactly the same as its execution on $\texttt{mapM}(m)$ after $\texttt{mapN}(n_m)$ and could not cover any changed CFG nodes or edges (because it does not even have a CFG path leading to the changed nodes or edges). Therefore, $\texttt{reach}(\texttt{mapN}(n_m), t)$ holds, and based on (the second disjunct of) `DReachabilityCheck` (Section 4.4.3.2), t would be selected in T_r. Contradiction.

In summary, for any mutant m of P', every test t in T'_c that is not selected in T_r has an equivalent execution on the corresponding mutant of P. In other words, ReMT only reuses results that are exactly the same on both versions, and is thus safe. □

4.5 Implementation

We built ReMT on top of Javalanche [122], a state-of-the-art tool for mutation testing of Java programs with JUnit tests. Javalanche allows efficient

mutant generation as well as efficient mutant execution. It uses a small set of sufficient mutation operators [96], namely replace numerical constant, negate jump condition, replace arithmetic operator, and omit method calls [122]. Javalanche manipulates Java bytecode directly using mutant schemata [131] to enable efficient mutant generation. For efficient mutant execution, Javalanche does not execute the tests that do not reach the mutated statement, and it executes mutants in parallel. It provides the `javalanche.stop.after.first.fail` configuration property to select partial or full mutation scenario.

Our ReMT implementation extends Javalanche with dangerous-edge reachability checking and mutation-specific test prioritization. For static analysis, our implementation uses the intra-procedural CFG analysis of the *Sofya tool* [71] to obtain basic intra-procedural CFG information and uses the *Eclipse JDT toolkit* [2] to obtain the inter-procedural information (method-overriding hierarchy, type-inheritance information, etc.) for inter-procedural CFG analysis. As a way to test our implementation, our experimental study verified that the incrementally collected mutation testing results by ReMT are the same as (non-incrementally collected) mutation testing results by Javalanche.

4.6 Experimental Study

ReMT aims to reduce the cost of mutation testing by utilizing the mutation testing results from a previous program version. To evaluate ReMT, we compare it with Javalanche [122], the state-of-the-art tool for mutation testing.

4.6.1 Research Questions

Our study addresses the following research questions:

- **RQ1:** How does ReMT compare with Javalanche, which does not use history information, in the full mutation testing scenario in terms of both efficiency and effectiveness?
- **RQ2:** How does ReMT compare with Javalanche in the partial mutation testing scenario under different original test-suite orders?
- **RQ3:** How does the mutation-specific test prioritization (MTP) further optimize ReMT in the partial mutation testing scenario?

4.6.2 Independent Variables

We used the following three independent variables (IVs):

IV1: Different Mutation Testing Techniques. We considered the following choices of mutation testing techniques: (1) Javalanche, (2) ReMT, and (3) ReMT+MTP.

IV2: Different Mutation Testing Scenarios. We considered two mutation testing scenarios for applying mutation testing: (1) full mutation testing and (2) partial mutation testing.

IV3: Different Test-Suite Orders. As the performance of all evaluated techniques under the partial mutation testing scenario depends on the test-suite orders, we used 20 randomized original test-suite orders for each studied

revision to evaluate the performance of each technique under that scenario.

4.6.3 Dependent Variables

Since we are concerned with the effectiveness as well as efficiency achieved by our ReMT technique, we used the following two dependent variables (DVs):

DV1: Number of Mutant-Test Executions. This variable denotes the total number of mutant-test pairs executed by the compared techniques.

DV2: Time Taken. This variable records the total time (including test execution time and technique overhead) taken by the compared techniques.

4.6.4 Subjects and Experimental Setup

We used the source code repositories of six open-source projects in various application domains. Table 4.4 summarizes the projects. The sizes of the studied projects range from 3.9K lines of code (LoC) (*JDepend*, with 2.7KLoC source code and 1.2KLoC test code) to 88.8KLoC (*Joda-Time*, with 32.9KLoC source code and 55.9KLoC test code). We applied our ReMT on five recent revisions of each project. We treated each commit involving source code or test code changes as a revision; for commits conducted within the same day, we merged them into one revision. Table 4.5 shows more details for each revision in the context of mutation testing: Column 1 names the studied revision; Column 2 shows the number of source/test files committed; Columns 3 and 4 show the number of tests and mutants; Column 5 shows the ratio of killed mutants to all mutants and the ratio of killed mutants to reached

Table 4.4: Subjects overview.

Projects	Description	Source+Test(LoC)
JDepend	Design quality metrics	2.7K+1.2K
TimeMoney	Time and money library	2.7K+3.1K
Barbecue	Bar-code creator	5.4K+3.3K
Jaxen	Java XPath library	14.0K+8.8K
Com-Lang	Java helper utilities	23.3K+32.5K
Joda-Time	Time library	32.9K+55.9K

mutants.

In this experimental study, for the full mutation testing scenario, both the input and output mutation matrices are full (no `U`), and for the partial mutation testing scenario, both the input and the output mutation matrices are partial (but with enough information to compute the mutation score). The experimental study was performed on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

4.6.5 Results and Analysis

4.6.5.1 RQ1: Full Mutation Testing Scenario

In Table 4.5, Column 6 shows the total possible number of mutant-test executions without any reduction techniques, i.e., the product of the numbers of tests and mutants. Columns 7-9 show the actual number of executions performed by Javalanche and ReMT, and the reduction in the number of executions by ReMT over Javalanche. First, we observe that both Javalanche and ReMT significantly reduce the number of executions from the total possible executions. For instance, for all five revisions of *Barbecue*, the total possible number of executions are more than 5 million, while both Javalanche and ReMT are able to reduce the number of executions to around or be-

low 0.02 million. Second, although the reductions of ReMT over Javalanche vary greatly across subject revisions, ReMT is able to further achieve reductions of more than 50% on the majority of all the revisions. Furthermore, ReMT is able to achieve reductions of more than 90% on 12 of the 30 studied revisions. For instance, on revision *TimeMoney*-4, ReMT is even able to identify that no executions are required to get the new mutation testing results. Manually inspecting the code changes in this revision, we found that the developers changed parts of two source files that cannot be reached by any tests, and thus the mutation testing results cannot be influenced. However, there are also revisions for which ReMT cannot achieve much reduction. For instance, on revision *Jaxen*-2, ReMT is able to achieve a reduction of only 0.08% over Javalanche. We looked into the revision history and found that the developers conducted an import patch across all methods of that `org.jaxen.saxpath.base.XPathLexer` class that is a central class used by nearly all the tests in the suite.

Columns 10-12 compare the actual tool time rather than the number of executions. Column 10 of Table 4.5 shows the mutation testing time taken by Javalanche. Column 11 shows the overall mutation testing time taken by ReMT, including the time taken by the preprocessing steps of ReMT (specifically by mutant mapping and dangerous-edge reachability analysis)[1]. Column 12 shows the reduction of costs by ReMT over Javalanche in terms of

[1]We do not explicitly measure the coverage preprocessing time because node coverage is already traced by Javalanche, and edge coverage is available for any system using regression test selection.

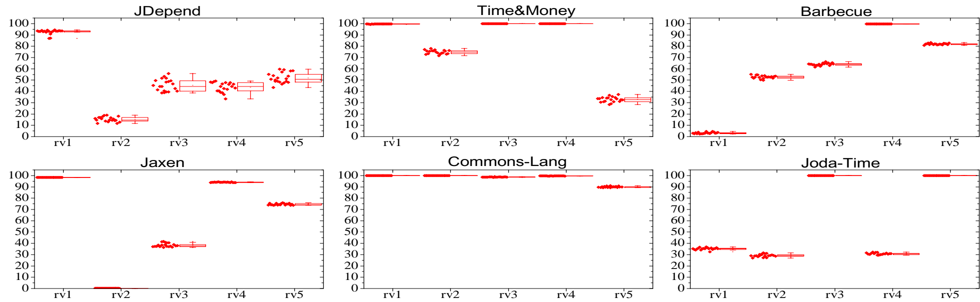
time. First, we observe that the reduction in terms of time does not directly match the reduction in terms of the number of executions; sometimes the reduction for time is lower (e.g., *JDepend*-1), and sometimes it is higher (e.g., *JDepend*-2). The likely reasons for this include the following: (1) the times for different executions vary significantly, (2) the reachability checking (lines 9-14 in Algorithm 2) needs extra time, and (3) Javalanche's parallel thread scheduling, database setup, and database access can influence the execution time. Second, we observe that our preprocessing step scales quite well: it takes at most 3 minutes and 33 seconds across all revisions (*Joda-Time*-1) and is negligible compared to the mutant-test execution time.

4.6.5.2 RQ2: Partial Mutation Testing Scenario

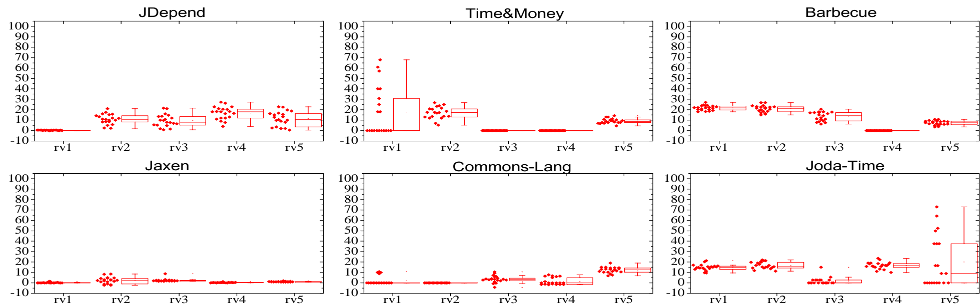
As different test-suite orders influence the performance of techniques under the partial mutation testing scenario, we evaluated the performance of ReMT and Javalanche under 20 different original test-suite orders. Figure 4.4(a) shows the reduction that ReMT achieves over Javalanche in terms of executions. In each plot, the horizontal axis shows different revisions of each subject, and the vertical axis shows the ratios of executions reduced by ReMT over Javalanche. Each box plot shows the mean (a dot in the box), median (a line in the box), upper/lower quartile, and max/min values for the reduction ratios achieved over 20 randomized original test-suite orders on each revision of each studied subject. The corresponding data dots are also shown to the left of each box plot. First, we observe that the reduction achieved by ReMT

Table 4.5: Experimental results of Javalanche and ReMT under the full mutation testing scenario.

Revision	Cs	Ts	Ms	Mutant Kill Rates(%)	Total Execs	Number of Executions			Time Taken		
						Javalanche	ReMT	Red.	Javalanche	ReMT (Cost)	Red.
JDepend-1	2	53	1,067	65.97/84.00	56,551	10,769	1,196	88.89%	00:05:45	00:02:04 (00:05	64.05%
JDepend-2	8	53	1,166	67.40/84.24	61,798	12,000	10,516	12.37%	00:06:14	00:02:21 (00:05	62.29%
JDepend-3	3	54	1,174	67.46/83.98	63,396	12,528	7,492	40.19%	00:06:00	00:02:11 (00:06	63.61%
JDepend-4	2	55	1,174	67.97/84.62	64,570	12,956	7,920	38.87%	00:06:04	00:02:07 (00:06	65.10%
JDepend-5	2	55	1,174	67.97/84.62	64,570	12,956	6,826	47.31%	00:06:06	00:02:40 (00:05	56.28%
TimeMoney-1	1	235	2,293	72.21/87.02	538,855	15,320	58	99.62%	00:09:08	00:02:11 (00:08	76.09%
TimeMoney-2	2	236	2,305	72.27/87.08	543,980	15,663	3,637	76.77%	00:09:19	00:02:23 (00:07	74.41%
TimeMoney-3	4	236	2,305	72.27/87.08	543,980	15,663	0	100.00%	00:09:18	00:02:12 (00:07	76.34%
TimeMoney-4	2	236	2,305	72.27/87.08	543,980	15,663	0	100.00%	00:09:18	00:02:12 (00:07	76.34%
TimeMoney-5	7	237	2,300	73.82/86.67	545,100	16,805	12,478	25.75%	00:09:22	00:07:30 (00:08	19.92%
Barbecue-1	3	154	36,419	2.75/68.39	5,608,526	21,267	20,852	1.95%	00:09:31	00:07:23 (00:15	22.41%
Barbecue-2	1	154	36,419	2.75/68.39	5,608,526	21,267	11,352	46.62%	00:09:32	00:05:57 (00:21	37.58%
Barbecue-3	1	154	36,419	2.75/68.39	5,608,526	21,267	8,850	58.39%	00:09:36	00:06:10 (00:12	35.76%
Barbecue-4	3	154	36,419	2.75/68.39	5,608,526	21,267	36	99.83%	00:09:47	00:04:21 (00:12	55.53%
Barbecue-5	1	154	36,419	2.75/68.39	5,608,526	21,267	4,715	77.82%	00:09:11	00:05:27 (00:12	40.65%
Jaxen-1	3	688	9,937	46.49/70.00	6,836,656	1,495,822	25,641	98.28%	01:06:35	00:16:05 (00:33	75.84%
Jaxen-2	5	689	9,876	46.69/70.53	6,804,564	1,489,630	1,488,367	0.08%	01:06:10	01:06:52 (00:42	-1.05%
Jaxen-3	3	690	9,881	46.71/70.53	6,817,890	1,493,341	998,045	33.16%	01:06:21	00:42:25 (00:23	36.07%
Jaxen-4	3	694	9,891	46.79/70.59	6,864,354	1,504,587	98,411	93.45%	01:06:50	00:21:07 (00:25	68.40%
Jaxen-5	2	695	9,901	46.84/70.63	6,881,195	1,508,683	430,521	71.46%	01:07:35	00:27:40 (00:25	59.06%
Com-Lang-1	5	1,689	19,747	65.63/86.21	33,352,683	93,423	46	99.95%	01:31:57	00:32:11 (01:42	64.99%
Com-Lang-2	8	1,691	19,747	65.64/86.21	33,392,177	93,425	19	99.97%	01:32:07	00:32:08 (01:32	65.11%
Com-Lang-3	2	1,691	19,747	65.69/86.25	33,392,177	93,430	1,124	98.79%	01:32:10	00:31:55 (01:32	65.37%
Com-Lang-4	2	1,691	19,747	65.68/86.23	33,392,177	93,430	352	99.62%	01:32:33	00:32:09 (01:31	65.26%
Com-Lang-5	3	1,692	19,747	65.68/86.24	33,411,924	93,450	10,915	88.31%	01:32:15	00:40:31 (01:31	56.07%
Joda-Time-1	2	3,818	24,175	65.09/85.41	92,300,150	1,064,395	776,299	27.06%	04:21:58	03:12:21 (03:33	26.57%
Joda-Time-2	2	3,828	24,190	66.47/87.19	92,599,320	1,076,987	865,922	19.59%	04:30:47	03:27:05 (03:23	23.52%
Joda-Time-3	3	3,829	24,219	66.43/87.09	92,734,551	1,077,851	619	99.94%	03:58:55	00:54:04 (01:56	77.37%
Joda-Time-4	7	3,832	24,236	66.71/87.44	92,872,352	1,077,757	795,152	26.22%	03:59:48	03:14:03 (03:29	19.07%
Joda-Time-5	1	3,834	24,236	66.41/87.05	92,920,824	1,078,573	1,443	99.86%	04:15:57	00:54:07 (01:44	78.85%



(a) Reduction of executions (%) achieved by ReMT over Javalanche with 20 randomization seeds for ordering original test suites.



(b) Reduction of executions (%) achieved by ReMT+MTP over ReMT with 20 randomization seeds for ordering original test suites.

Figure 4.4: Reduction of executions achieved by ReMT and MTP under the partial mutation testing scenario.

in the partial mutation testing scenario follows a similar trend as the reduction achieved in the full mutation testing scenario. In addition, the reductions achieved by ReMT over Javalanche under the partial mutation testing scenario are even slightly greater than under the full mutation testing scenario for 23 of the 30 revisions. Second, the reduction achieved by ReMT over Javalanche for each revision is not greatly influenced by different test suite orders: the standard deviation values for the reduction only range from 0 to 5.33. While the reduction ratios are not greatly influenced by test-suite orders, an interesting

finding is that the reduction ratios tend to be more stable when the reduction grows higher. For example, for all revisions with reduction ratios of more than 90%, different test-suite orders tend to have almost no impact at all on the reduction ratios.

4.6.5.3 RQ3: Mutation-Specific Test Prioritization

Figure 4.4(b) shows the further reduction of executions achieved by mutation-specific test prioritization (MTP) over ReMT using 20 randomized original test-suite orders for each revision. Each box plot has the same format as in Figure 4.4(a) except that the vertical axis represents the ratios of executions reduced by *ReMT+MTP over ReMT (and not overJavalanche)*. First, we observe that technique ReMT+MTP further achieves a reduction over ReMT on 26 of the 30 revisions. There are also four revisions where MTP does not make any further reductions over ReMT: *TimeMoney*-3, *TimeMoney*-4, *Barbecue*-4, and *Com-Lang*-2. The reason is that ReMT has already reduced the number of executions greatly, e.g., 0 executions for *TimeMoney*-3 and *TimeMoney*-4. Second, we observe that the reduction achieved by ReMT+MTP over ReMT for each revision can be greatly influenced by different original test-suite orders: the standard deviation values of the reduction range from 0 to 24.60, which contrasts with the reduction of ReMT over Javalanche. For example, the reductions achieved on *Joda-Time*-5 range from 0.00% to 72.97%. The reason is that MTP is just a reordering of all the tests identified by ReMT and can even execute *more* tests than the original test order executed by ReMT.

Although the prioritization is done for each mutant, the experimental study shows MTP is quite lightweight: on average, its total prioritization time for all mutants is less than 1sec. for the revisions of four projects (i.e., *JDepend*, *TimeMoney*, *Barbecue*, and *Com-Lang*), and is 2.43sec. and 3.58sec. for the revisions of *Joda-Time* and *Jaxen*, respectively.

4.7 Summary

This chapter makes the following contributions:

- **Regression Mutation Testing**. We introduce the idea of unifying regression testing with mutation testing—two well-researched methodologies that previous work has explored independently—to make mutation testing of evolving systems more efficient.

- **Technique**. We develop a core technique for regression mutation testing (ReMT) using dangerous-edge reachability analysis based on CFL reachability.

- **Optimization**. We introduce the idea of mutation-specific test prioritization (MTP) and present an MTP technique to optimize our core ReMT technique.

- **Implementation**. We implement ReMT and MTP on top of Javalanche [122], a recent mutation testing tool for Java programs with JUnit test suites.

- **Evaluation**. We present an empirical study on version repositories of six open-source Java programs between 3.9KLoC and 88.8KLoC. The results show that ReMT substantially reduce the costs of mutation testing on evolving systems.

Chapter 5

Test Prioritization and Reduction for Mutation Testing

The previous chapter presented our first unification of regression testing and mutation testing: using test selection to speed up mutation testing. In this chapter, we present our second approach for unification, which uses test prioritization and reduction to speed up mutation testing. Note that in contrast with regression test prioritization and reduction guided by test coverage, our approach mainly utilizes on-the-fly mutant killing history information to guide test prioritization and reduction for mutation testing. This chapter is based on our paper presented at the International Symposium on Software Testing and Analysis (ISSTA 2013) [153].

5.1 Background

The key insight into the effectiveness of mutation testing is that a test suite that kills a large number of mutants likely also finds a large number of real faults [11, 32, 46, 60]—even when the mutants are not the same as the real faults. This effectiveness of mutation testing relies on the ability to apply it on a *large* number of mutants. These mutants are generated systematically

using *mutation operators*, e.g., replace an integer constant with 0.

While mutation testing is very effective for evaluating test suite quality, it is also very *expensive* because it requires running many tests against many mutants. For each mutant that can be killed, we potentially run *several* tests that do not kill the mutant until we run one test that does kill the mutant. For each mutant that is not killed, we must run *every* test (that reaches the mutated statement). The cost of mutation testing can be measured in terms of the test-mutant pairs that are run. One way to reduce the cost is to reduce the number of mutants by selective mutation testing [16, 92, 96, 136, 149]. In contrast, this work reduces mutation testing cost in an orthogonal way: we aim to reduce the cost of executing tests for each mutant.

This chapter presents *Faster Mutation Testing* (FaMT) to reduce the cost of mutation testing. We build on the ideas of *test prioritization* [39, 116, 148] and *test reduction* [18, 22, 47, 50], which are central to regression testing [142]. Test prioritization has been applied to mutation testing by our previous work, ReMT [155], but ReMT is a specialized technique that (1) only works for evolving code and (2) requires old mutation testing results on previous versions to prioritize tests. In this chapter, we present the general FaMT approach to test prioritization for mutation testing which (1) works even for one code version and (2) does not require old mutation testing results. To the best of our knowledge, test reduction has not been previously used for mutation testing.

The goal of our test prioritization is to reorder the tests such that a

test that kills the mutant (when it can be killed) is run earlier. The goal of our test reduction is to run only a subset of tests on a mutant to determine that it is not killed if no test from this subset kills it. While test prioritization is *precise* in that it computes exactly the same mutation score as traditional mutation testing but does so faster, test reduction is *approximate* in that it provides an underapproximation for the mutation score (because a test that was not selected could kill a mutant even when no selected test kills it).

To compute the bounds of mutation testing cost, consider a program $\mathcal{P}$ with a set of mutants $\mathcal{M}$ and a test suite $\mathcal{T}$. Let the number of mutants killed by $\mathcal{T}$ be m_K and the number of mutants not killed by $\mathcal{T}$ be m_N; $|\mathcal{M}| = m_K + m_N$. Let the total number of test executions for the killed mutants be t_K ($t_K \geq m_K$) and the total number of test executions for the non-killed mutants be t_N. The total cost of mutation is $t_K + t_N$. Any *precise* (dynamic) technique that reduces this cost can only reduce the number of test executions to kill the mutants because for each mutant that is not killed, all tests (that reach the mutant) must be run. An *oracular* technique could kill each mutant (that can be killed) by running only one test per mutant, and therefore has cost $m_K + t_N$. An *approximate* technique could, in principle, have cost 0, by running none test-mutant pair, but would not kill any mutant. Thus, a goal to optimize mutation testing is to achieve higher precision with lower cost.

5.2 Example

This section illustrates our FaMT technique using a simple example with 4 mutants and 4 tests. Consider the following sample code snippet and its 4 mutants (m_1, m_2, m_3, m_4):

```
1  int abs(int x) {
2    int y = 0;
3    if (x < 0)
4      y =  x;
5    if (x < 0) {//m1:"if(y < 0)" m2:"if(x <= 0)"
6      return y;
7    } else {
8      return x;//m3:"return 0;" m4:"return  x;"
9    }
10 }
```

Note that each mutant is defined by exactly one change to the program, e.g., m_1 replaces the variable `x` with `y` at line 5. In the actual mutation testing, there would be many more mutants even for this simple code, but we use only 4 mutants for ease of exposition. Consider further the following 4 tests for this code:

```
1 assert abs(0) == 0;  // reach: m1, m2, m3, m4
2 assert abs(1) == 1;  // reach: m1, m2, m3, m4
3 assert abs( 1) == 1; // reach: m1, m2
4 assert abs( 4) == 4; // reach: m1, m2
```

The goal of mutation testing is to determine the mutation score for these 4 tests on these 4 mutants. A naïve approach would run all 4 tests on all 4 mutants to determine the mutation score. However, not all these 16 runs are necessary. Traditional mutation testing employs two optimizations. First, it is unnecessary to run tests on a mutant after it gets killed. Second, it is unnecessary to run tests that do not even reach the mutated statement when these tests are run on the original, unmutated program [6, 70, 122, 155]. For

Table 5.1: Traditional mutation testing

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_1	N	t_1	N	t_1	N	t_1	N	
t_2	N	t_2	N	t_2	K	t_2	K	
t_3	K	t_3	N					11
t_4	-	t_4	N					

Table 5.2: FaMT test prioritization

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_3	K	t_3	N	t_1	N	t_2	K	
t_4	-	t_4	N	t_2	K	t_1	-	
t_1	-	t_1	N					8
t_2	-	t_2	N					

Table 5.3: FaMT test reduction

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_3	K	t_3	N	t_1	N	t_2	K	
t_4	-	t_4	N	t_2	-	t_1	-	
t_1	-	t_1	-					5
t_2	-	t_2	-					

our example tests, the comments show this reachability information.

Table 5.1 shows the 11 test-mutant pairs that the traditional mutation runs with these two optimizations. The cells indicate the following: 'N' that the test was run but did not kill the mutant, 'K' that the test was run and did kill the mutant, and '-' that the test was not even run for that mutant. Note that t_3 and t_4 are not listed for m_3 and m_4 because they cannot reach the mutated statement, and t_4 is not run for m_1 because m_1 has been killed by t_3 before t_4.

FaMT improves on the optimized traditional mutation testing in two ways. First, FaMT uses test prioritization to reorder the tests that reach each mutant. (Section 5.3.3 presents the algorithm in detail.) Intuitively, our prioritization uses the dynamic information from test runs on the original program (recall that the tests are already run to find the reachability information) and on the history of test runs on other mutants which are tested before the current

mutant.

Table 5.2 shows the 8 test-mutant pairs that FaMT runs for our example. For each mutant, the table also shows how FaMT orders the tests for that mutant and the result of test runs. Note that FaMT can run the tests in different orders for different mutants. For m_1, FaMT orders t_3 and t_4 before t_1 and t_2 because t_3 and t_4 execute more instructions before reaching the mutated statement on the original program. (Section 5.3 describes the rationale for this choice and the additional information that FaMT uses.) For m_4, FaMT orders t_2 before t_1 because the execution history before m_4 (i.e., from m_1 to m_3) shows that t_2 is a "better killer"; t_2 killed m_3, whereas t_1 did not kill any mutant on which it was run before executing on m_4. In sum, the reordering that FaMT makes allows it to run only 8 test-mutant pairs (i.e., a reduction of 27.3%) and still produce the precise mutation score (3 of 4 mutants are killed).

Second, FaMT can further use test reduction to reduce the number of test-mutant pairs run. Intuitively, the reduction runs only a subset of the tests that reach a mutant; if none of these tests kills the mutant, FaMT considers that the mutant cannot be killed by the given test suite. (Section 5.3.4 discusses reduction techniques that FaMT can use.) To illustrate, consider that FaMT runs at most 50% of the tests that reach the mutant.

Table 5.3 shows the 5 test-mutant pairs that FaMT runs for our example. Note that different from FaMT prioritization, after t_3 and t_4 are executed on m_2, FaMT directly predicts that m_2 cannot be killed because 50% of the tests that reach m_2 have been executed. However, test reduction is *approxi-*

mate and provides an underapproximation for the mutation score because a test that was not selected to be run could kill a mutant even when no selected test kills it. For example, after t_1 is executed on m_3, t_2 will not be executed on m_3. Therefore, FaMT reduction imprecisely predicts that m_3 cannot be killed while actually it can be killed by t_1. In brief, using reduction allows FaMT to run only 5 test-mutant pairs (i.e., a reduction of 54.5%) but produces an imprecise mutation score (2/4 as opposed to the actual score of 3/4). Please note that our experimental study demonstrates that the underapproximations for real-world programs are much smaller than this example.

5.3 Approach

This section presents our FaMT approach for faster mutation testing. We first describe the basics of our approach, including initial test ordering (Section 5.3.1) and adaptive test ordering (Section 5.3.2). We then present test prioritization that FaMT performs to more quickly compute the precise mutation score (Section 5.3.3), and test reduction that FaMT performs to more quickly compute an approximate mutation score (Section 5.3.4). Recall that the cost of mutation testing has two key elements—running some tests for killed mutants and running every test for non-killed mutants. While FaMT prioritization addresses only the first element and calculates a precise mutation score, FaMT reduction addresses both of these elements but calculates an approximate mutation score.

5.3.1 Coverage-Based Initial Test Ordering

For each mutant m, FaMT first calculates the initial priority values of the tests that execute the mutated statement using test coverage on the unmutated program version. This calculation uses two basic heuristics: (1) tests that execute the mutated statement more times have a higher probability to kill the mutant [155], and (2) tests that execute the mutated statement more closely to the test exit statement have a higher probability to propagate the mutated state to the end and kill the mutant.[1] We also use a third heuristic that combines these two.

Let t be a test for mutant m. Our first heuristic calculates the initial priority value of t for m as:

$$\textsc{c}_1(t, m) = \textsc{CovNum}(t, stmt_m) \tag{5.1}$$

where $\textsc{CovNum}(t, stmt_m)$ denotes the number of times that t covers $stmt_m$ which is the mutated statement of m.

Our second heuristic calculates the initial priority value of t for m using the ratio of the number of statements executed by t before the *first* execution of $stmt_m$ to the number of all statements executed by t:

$$\textsc{c}_2(t, m) = \frac{\textsc{CovBefore}(t, stmt_m)}{\textsc{CovBefore}(t, stmt_m) + \textsc{CovAfter}(t, stmt_m)} \tag{5.2}$$

where $\textsc{CovBefore}(t, stmt_m)$ denotes the number of unique statements executed by t before the first execution of the mutated statement $stmt_m$, and $\textsc{CovAfter}(t, stmt_m)$

[1] Note that a test can cover the mutated statement multiple times; we measure the distance from the *first* execution of the mutated statement to the test exit statement.

denotes the number of unique statements executed by t after the first execution of $stmt_m$. The higher the value is, the closer $stmt_m$ may be to the end of t.

Our third heuristic combines the first two and calculates the initial priority value of t for m:

$$\textsc{c}_3(t, m) = \frac{\textsc{CovNum}(t, stmt_m) \times \textsc{CovBefore}(t, stmt_m)}{\textsc{CovBefore}(t, stmt_m) + \textsc{CovAfter}(t, stmt_m)} \tag{5.3}$$

5.3.2 Power-Based Adaptive Test Ordering

During the execution of the tests for a mutant, FaMT also collects on-the-fly history information to adaptively update the test execution order. The basic intuition is that a test which killed more mutants that are *close* to the current mutant has a higher likelihood to kill the current mutant. We call this likelihood of a test to kill the current mutant m as the *power* of a test with respect to m. Formally, we denote the mutation testing results as a matrix $\textsc{Matrix}$, where each cell $\textsc{Matrix}(t, m)$ denotes the execution result of t on m: `K` denotes that m is killed by t, and `N` denotes that m is executed but not killed by t. $\textsc{Matrix}$ is initially empty, and eventually filled with `N` and `K`. We define the power of test t with respect to m as the ratio of the number of mutants in m's neighborhood (denoted as $\mathcal{N}_m$, and defined below) which are killed by t to the number of all those in the neighborhood that have been executed by t (whether or not they are killed by t):

$$\textsc{p}_1(t, m) = \frac{|\{m' \in \mathcal{N}_m | \textsc{Matrix}(t, m') = \texttt{K}\}|}{|\{m' \in \mathcal{N}_m | \textsc{Matrix}(t, m') \in \{\texttt{K}, \texttt{N}\}\}|} \tag{5.4}$$

Intuitively, the higher the ratio, the higher the likelihood that t kills m. Note that the history information used by FaMT is *not* the mutation testing information from *previous program versions*. Instead, it is accumulating execution history of tests on other mutants that have been executed before the current mutants in the same mutation testing task.

Equation (5.4) calculates the power of a test by taking into account all the mutants which are in $\mathcal{N}_m$ and executed by t. However, the mutants that cannot be killed by any tests may unnecessarily lower the power of a test. Therefore, we propose another formula to calculate the power of a test by excluding the mutants that have not been killed by any test yet:

$$\text{P}_2(t, m) = \frac{|\{m' \in \mathcal{N}_m | \textsc{Matrix}(t, m') = \texttt{K}\}|}{|m' \in \mathcal{N}_m | \{\texttt{K}(m') \wedge \textsc{Matrix}(t, m') \in \{\texttt{K}, \texttt{N}\}\}|} \tag{5.5}$$

where $\texttt{K}(m')$ denotes whether m' has been killed by a test, i.e., $\texttt{K}(m') \Leftrightarrow \exists t, \textsc{Matrix}(t, m') = \texttt{K}$.

While we believe there are various ways to define the neighborhood among mutants, here we consider the mutants that share common program locations as neighbors. In particular, we define four levels of neighborhood, each of which can be used for calculating the power of a test with respect to a mutant:

- **Statement-level history**: FaMT groups all the mutants that occur on the same statement with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | stmt_{m'} = stmt_m\}$, where $stmt_m$ denotes the statement on which m occurs.

- **Method-level history**: FaMT groups all the mutants within the same method with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | meth_{m'} = meth_m\}$, where $meth_m$ denotes the source method in which m occurs.

- **Class-level history**: FaMT groups all the mutants that occur in the same class with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | clas_{m'} = clas_m\}$, where $clas_m$ denotes the source class in which m occurs.

- **Global history**: We group all the mutants as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | m' \in \mathcal{M}\}$, where $\mathcal{M}$ denotes all the mutants for the program under test.

Different levels of neighborhood enable FaMT to use different levels of history information. For each history level, FaMT utilizes the history information of a test t on the mutants that occur in the same neighborhood (e.g., in the same class) with the current mutant m to calculate the likelihood that t kills m. Statement-level history calculates the test power based on mutants that are on the same statement because those mutants tend to perform similarly when being tested. However, the number of mutants that are on the same statement is usually too small for sampling. In contrast, global-level history records the test power for the entire program and may be imprecise for specific mutants, but the number of mutants is sufficiently large. Therefore, we use all four levels of history information to investigate their impacts.

Algorithm 3: FaMT Prioritization Algorithm

```
   Input: Program 𝒫, mutants ℳ, test suite 𝒯
   Output: MATRIX
1  begin
2    Initialize MATRIX as empty
     // Collect coverage information when executing 𝒯 on 𝒫
3    COVNUM, COVBEFORE, COVAFTER ← COVCOLLECT(𝒯, 𝒫)
4    for m ∈ ℳ do
       // Detect tests that execute the mutated statement on 𝒫
5      𝒯_m ← {t ∈ 𝒯 | COVNUM(t, stat_m) > 0}
6      for t : 𝒯_m do
         // Note that the initial priority calculation is not updated during
            mutation testing
7        Calculate C(t, m) according to Section 5.3.1
         // Note that the power calculation is continuously updated during
            mutation testing
8        Calculate P(t, m) according to Section 5.3.2
       // Reorder 𝒯_m based on the initial priority values, C
9      𝒯'_m ← REORDER(𝒯_m, C)
       // Split 𝒯'_m into two lists by comparing the power values, P, with
          Threshold
10     𝒯_1, 𝒯_2 ← PARTITION(𝒯'_m, P, Threshold)
       // Iterate over the test list by concatenating 𝒯_1 and 𝒯_2
11     for t : 𝒯_1 ⊕ 𝒯_2 do
12       MATRIX(t, m) ← EXECUTE(t, m) // N or K
         // If mutant m is killed, continue to next mutant
13       if MATRIX(t, m) = K then break
     // Return the final mutation testing matrix
14   return MATRIX
```

5.3.3 Test prioritization

The goal of our test prioritization technique is to reorder the tests such that a test that kills the mutant (when it can be killed) is run earlier than by simply following the default or random order of tests. Algorithm 3 gives the pseudo-code for our technique. The algorithm employs a family of ordering functions. These functions are based on coverage information of tests (Section 5.3.1) and on the accumulating execution history of tests (Section 5.3.2).

The algorithm takes program $\mathcal{P}$, its mutant set $\mathcal{M}$, and test suite $\mathcal{T}$ as inputs, and returns the mutation testing matrix MATRIX as output. Line

2 initializes MATRIX as empty. Line 3 collects coverage information which is used during later steps. Lines 4-13 iterate over the mutant set to determine whether each mutant is killed. During each iteration, Line 5 identifies the set of tests $\mathcal{T}_m$ that reach the mutated statement of current mutant m on the unmutated program, because the tests which do not reach the mutated statement cannot kill the mutant $[6, 70, 122, 155]$. Lines 6-8 iterate over all the tests in $\mathcal{T}_m$ and calculate the initial priority (Section 5.3.1) as well as power (Section 5.3.2) for each test with respect to m. Note that the initial test priorities are fixed during the process of mutation testing, because they are based on the coverage information of the tests on the unmutated program. However, the test powers are continuously updated during the mutation testing process: the more mutants in the neighborhood of m executed, the higher the accuracy of the power values.

Line 9 reorders $\mathcal{T}_m$ according to the initial priorities of tests. Line 10 then partitions the reordered list into two sublists, $\mathcal{T}_1$ and $\mathcal{T}_2$, based on the power of a test: if the power is less than the `Threshold`, FaMT puts the test into $\mathcal{T}_2$; otherwise, FaMT puts the test into $\mathcal{T}_1$. Note that each sublist is still ordered by initial test priorities. Lines 11-13 concatenate $\mathcal{T}_1$ with $\mathcal{T}_2$ and iterate over the concatenated list. Line 12 executes mutant m on test t and puts its execution result into the resulting MATRIX: if t kills m, the execution result is `K`; otherwise, the result is `N`. Line 13 terminates the execution for current mutant and continues to the next mutant if the current mutant is killed. Finally, Line 14 returns the mutation testing matrix MATRIX as output

and terminates the algorithm.

5.3.4 Test reduction

Our test prioritization can reduce the number of executions to kill mutants, but for the mutants that cannot be killed, the test prioritization cannot help. The goal of our test reduction is to run only a subset of tests on a mutant to determine that it is not killed if no test from this subset kills it. In this way, we can reduce the number of executions for all the mutants, regardless of whether they are killed or not. Note that this reduction may cause the mutation testing result to be approximate because some mutant may be mistakenly predicted as not killable due to some tests that kill it being omitted. Therefore, this algorithm needs to be carefully evaluated through an empirical study.

The basic intuition of our test reduction is that if those tests with higher likelihood to kill a mutant cannot kill the mutant, the remaining tests will have little chance to kill the mutant. Recall that our test prioritization also executes first the tests that have higher likelihood to kill mutants. Therefore, we build our test reduction algorithm directly on our test prioritization algorithm (Algorithm 3). Our reduction modifies Line 11 of Algorithm 3. While our prioritization algorithm always concatenates the entire $\mathcal{T}_1$ and $\mathcal{T}_2$, our reduction algorithm only concatenates $\mathcal{T}_1$ with a prefix of $\mathcal{T}_2$. More specifically, we change Line 11 into the following line to form our reduction algorithm:

$$\textbf{for } t : \mathcal{T}_1 \oplus \textsc{PreFix}(\mathcal{T}_2, \textsc{Max}(0, (|\mathcal{T}_1 \oplus \mathcal{T}_2|) \times \texttt{MinRatio} - |\mathcal{T}_1|))$$

Table 5.4: Subjects

Subject	Version	Size	#Tests	#Mutants(KillRates)
TimeMoney	r207	2681	236	2304 (72.35/87.14)
Jaxen	r1346	13946	690	9880 (46.72/70.54)
Xml-Sec	v3.0	19796	84	9693 (26.41/70.93)
Com-Lang	r1040879	23355	1691	19746 (65.68/86.24)
JDepend	v2.9	2721	55	1173 (68.03/84.62)
Joda-Time	r1604	32892	3818	24174 (66.45/87.16)
JMeter	v1.0	36910	60	21896 (9.24/28.34)
Mime4J	v0.50	6954	120	19111 (23.10/63.39)
Barbecue	r87	5391	154	36418 (2.75/68.40)

where $\textsc{PreFix}(\mathcal{T}_2, x)$ returns the sublist which contains the first x tests in $\mathcal{T}_2$, and $\textsc{Max}(x, y)$ returns the larger of x or y. The reduction algorithm concatenates $\mathcal{T}_1$ with a prefix of $\mathcal{T}_2$ such that ratio of the concatenated list's length to the length of $\mathcal{T}_1 \oplus \mathcal{T}_2$ is at least `MinRatio`. Note that if the length of $\mathcal{T}_1$ is already larger than or equal to $(|\mathcal{T}_1 \oplus \mathcal{T}_2|) \times$`MinRatio`, no tests from $\mathcal{T}_2$ will be executed.

5.4 Experimental Study

FaMT aims to reduce the cost of mutation testing by prioritizing and reducing the tests that need to be executed for each mutant. To evaluate FaMT, we implement FaMT on top of Javalanche [122], a state-of-the-art mutation testing tool for Java.

5.4.1 Research Questions

Our experimental study addresses these research questions:

- **RQ1:** How does FaMT prioritization reduce the number of executions?
- **RQ2:** How does FaMT reduction reduce the number of executions and how it approximates the mutant killing ratio?

- **RQ3:** How does FaMT compare with regression test prioritization and reduction in the mutation testing scenario? (While regression test prioritization and reduction are not originally designed for mutation testing, they have a straightforward application to it—to compare with FaMT, we apply regression test prioritization and reduction to mutation testing by using the coverage information of the original program to uniformly prioritize and reduce tests across all the mutants.)

- **RQ4:** What are the runtime overheads for both the test prioritization and test reduction of FaMT?

5.4.2 Independent Variables

We used the following independent variables (IVs):

IV1: Different Initial Orderings. We considered different test ordering randomizations for each mutant (*Org*) and all our three coverage-based orderings (C_1, C_2, and C_3; Section 5.3.1).

IV2: Different Test Power Formulas. We considered both choices of test power formulas presented in Section 5.3.2: (1) using history of all neighbor mutants and (2) using history of only killed neighbor mutants. We denote them as P_1 and P_2, respectively.

IV3: Different History Information Levels. We considered all four levels of history information presented in Section 5.3.2: (1) statement level, (2) method level, (3) class level, and (4) global level. We denote them as *Stat*,

Meth, *Clas*, and *Glob*, respectively.

IV4: Different Thresholds. We considered 11 `Threshold` values for Algorithm 3, ranging from 0.0 to 1.0 with increments of 0.1.

IV5: Different MinRatios. We considered 11 `MinRatio` values from Section 5.3.4, ranging from 0.0 to 1.0 with increments of 0.1.

IV6: Different Regression Testing Techniques. We considered the widely used *total* and *additional* regression test prioritization techniques using statement coverage [39, 116], and the widely used *greedy* regression test reduction technique using statement coverage [154], to evaluate their effectiveness for RQ3.

5.4.3 Dependent Variables

To evaluate the effectiveness and the efficiency of FaMT, we used the following three dependent variables (DVs):

DV1: Execution Reduction Ratio. This variable denotes the ratio of test-mutant executions reduced by FaMT prioritization or reduction to the number of all test-mutant executions that reach mutants.

DV2: Error Rate. This variable shows the ratio of mutants that are mistakenly predicted as unkillable by FaMT reduction, i.e., $err = \frac{|\mathcal{M}_e|}{|\mathcal{M}_r|}$, where $\mathcal{M}_e$ denotes the set of mistakenly predicted mutants and $\mathcal{M}_r$ denotes all the reached mutants.

DV3: Run Time Overhead. This variable records the runtime overheads

incurred by FaMT prioritization or reduction. Specifically, we recorded all the extra setup costs for FaMT prioritization/reduction in comparison with Javalanche, including calculating the coverage-based heuristic and test power information, as well as prioritizing/reducing tests based on them.

5.4.4 Subjects and Experimental Setup

We evaluated FaMT using nine open-source projects which come from various application domains and have been widely used for mutation testing and regression testing research [121, 122, 148, 155]. Table 5.4 summarizes the projects. The sizes of the studied projects range from 2.6K lines of code (LoC) to 36.9KLoC (excluding blank lines and test code). Column 4 shows the number of tests for each subject. In the last column of Table 5.4, we show the number of all generated mutants, the ratio (%) of killed mutants to all the mutants, and the ratio (%) of killed mutants to the reached mutants.

We evaluate all the FaMT techniques on all subjects:

For FaMT prioritization, we studied all the combinations of 4 initial orderings, 2 choices of power calculation, 4 levels of history information, and 11 `Threshold` values, i.e., 4*2*4*11=352 prioritization variant techniques.

For FaMT reduction, we studied all the combinations of 4 initial orderings, 2 choices of power calculation, 4 levels of history information, 11 `Threshold` values, and 11 `MinRatio` values, i.e., 4*2*4*11*11=3872 reduction variant techniques.

As the accumulated history varies for different orders of mutant exe-

cution, we randomize the execution order for mutants and apply each FaMT prioritization or reduction technique on each subject for 20 times to evaluate its effectiveness as well as the stability. We also applied all other compared techniques for 20 times on each subject. The experiments were performed on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

5.4.5 Result Analysis

All the detailed experimental data can be found online [40].

5.4.5.1 RQ1: FaMT Test Prioritization

Evaluation with default threshold. We applied FaMT prioritization techniques with the default `Threshold` of 0.3 on all the subjects. Table 5.5 shows the detailed experimental results for FaMT prioritization techniques using the history of all neighbor mutants (P_1). Column 1 ($A.$) and Column 2 ($I.$) show the levels of *adaptive* history information and the *initial* test orderings used. Columns 3-20 present the ratios of test executions for killed mutants reduced by the studied techniques (both mean values and standard deviations over 20 runs) compared with different randomized test orders for each mutant (DR). Column 21 presents the total execution reduction ratios (the ratio of the sum of all reduced executions to the sum of all executions over all subjects) by each technique. Similarly, Table 5.6 shows the experimental results of FaMT prioritization using the history of only killed neighbor mutants (P_2). We also

compare the baseline *DR* with a more basic random technique, *UR*, which randomizes the entire original test suite and then reorders the tests for each mutant according to their ordering in the randomized original suite (first row in Table 5.5). The key difference between *UR* and *DR* is that *UR* uses the same random ordering for all mutants whereas *DR* uses different random orderings for different mutants. Our findings are as follows.

First, FaMT prioritization techniques that embody history information perform better than techniques without history information. For example, *UR*, C_1, C_2 and C_3 without history information reduce the number of executions by -4.6% to 12.8% in total (compared with *DR*). In contrast, all the techniques with history information can effectively reduce the number of executions by 5.4% (C_1 with global-level history and P_1 formula) to 46.2% (C_3 with class-level history and P_2 formula) in total. Another interesting finding is that history power information can even boost the *Org* test order to achieve high reduction ratios. For example, when using the class level history and P_2, even *Org* can reduce the number of executions by 42.2% in total. We also performed a statistical test to confirm the effectiveness of FaMT. Specifically, the Fisher's LSD test [134] shows that all FaMT techniques in Table 5.6, except the first two, outperform the *Org* random technique at the significance level of 0.05.

Second, for all levels of history information, using P_2 performs better than using P_1 in reducing executions. For example, for the method level history information, techniques using P_2 reduce the number of executions by 39.7%

Table 5.5: Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of all neighbor mutants (P_1)

A.	I.	*TimeMoney* Red.	*TimeMoney* Dev.	*Jaxen* Red.	*Jaxen* Dev.	*Xml-Sec* Red.	*Xml-Sec* Dev.	*Com-Lang* Red.	*Com-Lang* Dev.	*JDepend* Red.	*JDepend* Dev.	*Joda-Time* Red.	*Joda-Time* Dev.	*JMeter* Red.	*JMeter* Dev.	*Mime4J* Red.	*Mime4J* Dev.	*Barbecue* Red.	*Barbecue* Dev.	Total Red.
-	*UR*	0.8	9.39	-18.4	61.81	-2.2	18.24	-0.6	3.47	-2.5	21.75	-0.2	7.32	-0.2	3.75	3.9	8.26	2.9	10.15	-1.7
-	*Org*	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
-	C_1	4.9	4.71	-45.8	60.44	2.0	5.87	2.7	1.43	-3.0	8.12	16.5	2.98	0.6	1.61	6.7	4.50	9.8	5.67	-4.6
-	C_2	0.0	6.19	26.3	27.97	-15.2	11.78	-7.5	1.75	-11.5	10.33	-0.0	4.24	-1.8	2.16	-11.0	4.98	-14.7	7.11	12.2
-	C_3	1.8	4.82	2.4	35.58	3.7	9.61	-1.1	1.60	7.0	8.31	19.5	3.39	-0.6	2.14	7.3	4.07	4.3	7.29	12.8
S	*Org*	14.2	3.28	15.7	5.85	19.1	6.40	8.1	0.79	17.4	2.32	14.5	1.39	4.1	0.82	16.6	1.81	25.0	2.72	13.7
t	C_1	18.4	5.06	-27.4	55.48	21.9	7.54	10.2	1.33	14.8	6.60	25.0	3.35	4.8	1.68	20.0	3.67	28.2	4.77	7.6
a	C_2	13.3	5.30	53.5	17.16	11.5	10.17	3.2	1.49	12.5	7.47	16.9	3.65	2.6	2.09	11.1	4.18	18.0	5.36	31.6
t	C_3	16.8	4.94	20.5	30.04	22.5	8.99	8.2	1.42	21.0	6.77	29.3	3.42	3.8	2.12	20.3	3.55	26.2	5.59	25.5
M	*Org*	18.0	4.19	20.1	8.49	26.9	8.22	9.7	0.91	22.5	3.17	23.1	1.65	6.4	1.55	18.9	2.86	32.6	3.69	18.9
e	C_1	20.7	5.03	-22.3	53.64	28.4	8.61	11.7	1.23	22.2	6.52	29.2	3.06	7.4	1.92	20.6	3.76	35.2	4.30	11.4
t	C_2	17.5	5.23	58.9	15.54	21.7	9.97	6.2	1.46	20.2	6.80	25.0	3.17	5.1	2.02	15.2	4.39	27.0	4.74	37.4
h	C_3	19.8	5.15	24.2	29.30	28.6	9.10	9.9	1.22	27.3	6.57	33.3	3.11	6.6	2.12	20.9	3.49	33.7	4.50	28.8
C	*Org*	17.4	3.57	14.1	5.33	27.2	9.42	8.5	0.97	22.7	8.36	28.2	2.51	6.4	1.47	17.1	3.23	25.9	3.51	18.4
l	C_1	18.3	4.82	-28.1	56.30	28.9	9.06	10.4	1.28	19.4	11.54	34.3	3.28	7.5	1.94	16.6	4.70	32.6	3.76	10.8
a	C_2	18.5	5.29	55.3	18.28	24.6	9.77	6.5	1.44	20.4	8.93	28.0	3.62	5.4	2.03	14.3	4.81	21.2	6.70	37.3
s	C_3	18.3	4.84	19.2	30.76	29.4	9.45	8.6	1.27	16.7	11.29	36.8	2.86	6.6	2.10	17.3	4.13	31.1	4.78	27.8
G	*Org*	14.1	3.04	2.2	1.49	23.2	8.32	7.7	1.17	23.3	5.99	29.4	2.67	4.7	1.61	15.1	4.07	18.8	5.76	14.1
l	C_1	15.2	4.91	-42.6	59.75	24.4	8.92	10.2	1.30	21.2	10.08	34.8	3.31	6.5	1.95	12.1	5.76	24.6	5.20	5.4
o	C_2	15.5	5.88	29.2	27.79	20.1	10.06	5.8	1.41	17.9	10.11	31.0	3.88	5.3	2.03	12.6	6.40	11.6	8.12	28.9
b	C_3	13.9	4.84	4.6	34.98	26.1	8.43	8.5	1.36	19.4	7.78	35.6	3.38	6.1	2.56	13.8	5.62	21.6	5.79	22.0

Table 5.6: Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of only killed neighbor mutants (P_2)

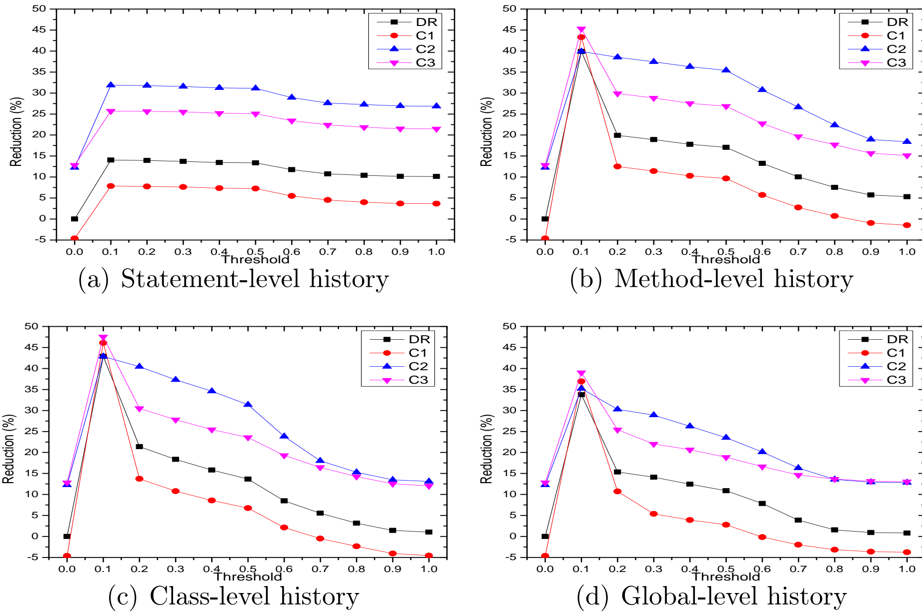
A.	I.	*TimeMoney* Red.	*TimeMoney* Dev.	*Jaxen* Red.	*Jaxen* Dev.	*Xml-Sec* Red.	*Xml-Sec* Dev.	*Com-Lang* Red.	*Com-Lang* Dev.	*JDepend* Red.	*JDepend* Dev.	*Joda-Time* Red.	*Joda-Time* Dev.	*JMeter* Red.	*JMeter* Dev.	*Mime4J* Red.	*Mime4J* Dev.	*Barbecue* Red.	*Barbecue* Dev.	Total Red.
S	*Org*	14.5	3.21	16.0	6.03	19.4	6.57	8.3	0.70	17.4	2.35	14.7	1.36	4.2	0.86	17.0	1.78	25.2	2.67	13.9
t	C_1	18.8	4.89	-27.3	55.32	22.4	7.57	10.4	1.32	14.8	6.56	24.9	3.26	4.9	1.73	20.0	3.66	28.3	4.72	7.6
a	C_2	13.7	5.38	53.3	17.30	11.9	10.12	3.3	1.49	12.5	7.41	17.1	3.64	2.6	2.11	11.4	4.21	18.1	5.34	31.6
t	C_3	17.2	4.91	20.5	30.24	23.0	9.00	8.4	1.39	21.0	6.85	29.3	3.44	3.8	2.14	20.4	3.56	26.2	5.58	25.5
M	*Org*	17.9	3.93	65.3	12.99	27.8	8.63	9.9	0.89	22.6	3.39	23.4	2.42	5.9	1.32	21.5	2.79	35.7	3.13	40.1
e	C_1	21.1	4.71	68.0	11.53	29.5	8.92	12.0	1.31	22.7	5.92	29.9	3.08	7.0	1.87	23.7	3.54	37.2	4.31	43.8
t	C_2	17.4	5.03	63.7	13.12	23.7	9.87	6.3	1.34	20.1	6.60	26.1	3.49	4.3	2.08	17.7	3.98	33.7	4.25	39.7
h	C_3	20.0	4.87	69.0	11.45	29.8	8.78	9.7	1.33	28.1	6.10	34.4	3.23	5.8	2.07	23.7	3.67	38.7	4.70	45.5
C	*Org*	15.6	3.63	68.4	12.84	27.5	8.92	8.3	0.89	23.3	6.04	26.8	2.21	6.0	1.45	20.5	2.89	36.3	3.41	42.2
l	C_1	17.1	4.80	66.3	13.33	28.4	9.25	10.7	1.46	24.7	7.84	33.3	2.76	7.0	1.86	23.9	4.15	38.9	4.46	44.3
a	C_2	16.5	5.35	68.1	12.62	26.1	9.47	5.7	1.45	23.2	7.89	30.2	3.51	4.3	1.95	15.6	3.87	33.1	4.74	42.6
s	C_3	16.9	4.85	67.4	13.01	28.8	9.47	8.8	1.42	29.4	5.55	38.3	2.66	5.4	2.03	23.9	3.61	38.6	4.80	46.2
G	*Org*	10.8	2.56	65.5	12.78	17.1	6.80	7.3	0.98	19.5	5.57	30.1	2.55	4.2	1.33	13.1	2.40	31.5	2.89	41.1
l	C_1	13.7	4.53	59.5	15.64	19.0	7.99	10.0	1.50	19.8	7.50	36.5	3.36	5.7	1.94	19.9	4.11	34.8	5.01	42.2
o	C_2	12.6	5.34	68.8	12.65	15.7	10.01	4.2	1.52	19.5	8.12	33.3	3.08	2.7	2.08	6.5	5.22	28.0	5.74	42.7
b	C_3	12.4	4.43	62.7	15.07	21.9	8.30	7.9	1.47	27.5	4.76	39.5	2.54	4.1	2.24	19.2	4.25	33.9	5.09	44.2

to 45.5%, while techniques using P_1 only reduce the number of executions by 11.4% to 37.4%. The reason is that the non-killable mutants may unnecessarily lower the power of tests and thus delay the execution of some *good* tests. The only exceptions are the techniques using statement-level history.

Third, for both P_1 and P_2, the techniques using the method or class level history information tend to perform the best. For example, FaMT techniques using method level history and P_1 formula reduce the number of executions by 11.4% to 37.4%, and FaMT techniques using class level history and P_2 formula reduce the number of executions by 42.2% to 46.2%. Interestingly, for those techniques, the best initial ordering within the same level of history information depends on using P_1 or P_2. For example, when using P_1, C_2 performs the best. On the contrary, when using P_2, C_3 performs the best.

Effectiveness trends when using various thresholds. Figure 5.1 illustrates the trends for FaMT prioritization techniques with P_1 formula using different thresholds from 0.0 to 1.0. The four plots are for FaMT techniques using statement, method, class, and global levels of history, respectively. In each plot, each line represents using the initial ordering *Org*, C_1, C_2, or C_3. Similarly, Figure 5.2 illustrates the trends for FaMT prioritization techniques with P_2 formula using different thresholds.

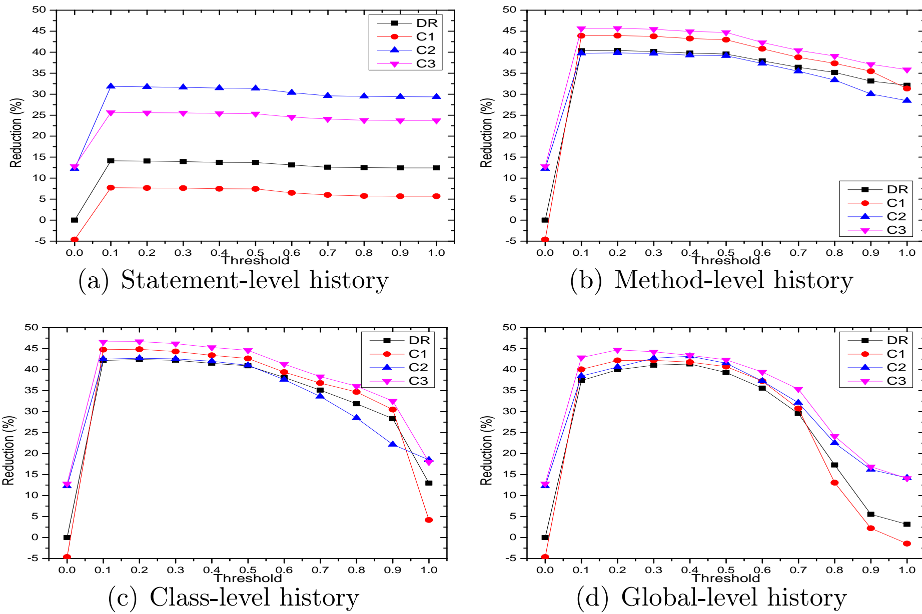
First, when `Threshold`=0.1, almost all FaMT techniques achieve highest reduction ratios. For example, techniques using class-level history and P_1 formula reduce executions by 42.86% to 47.52%, and techniques using class-level history and P_2 formula reduce executions by 42.17% to 46.63%. The only



(a) Statement-level history (b) Method-level history

(c) Class-level history (d) Global-level history

Figure 5.1: Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_1 formula



(a) Statement-level history (b) Method-level history

(c) Class-level history (d) Global-level history

Figure 5.2: Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_2 formula

Table 5.7: Execution reduction (%) for killed/all mutants by theoretical techniques and an FaMT prioritization technique.

Subject	Reduction for Killed Mutants			Reduction for All Mutants		
	T-Worst	T-Best	FaMT	T-Worst	T-Best	FaMT
TimeMoney	-115.6	34.3	16.9	-66.3	19.8	9.8
Jaxen	-505.7	90.4	67.4	-27.3	5.9	4.6
Xml-Sec	-122.9	41.3	28.8	-28.2	9.8	6.9
Com-Lang	-94.3	24.4	8.8	-56.4	14.6	5.3
JDepend	-179.8	47.7	29.4	-75.6	20.3	12.5
Joda-Time	-981.8	66.6	38.3	-459.3	31.2	18.0
JMeter	-41.8	14.9	5.4	-11.2	4.0	1.5
Mime4J	-131.1	41.4	23.9	-41.0	13.0	7.5
Barbecue	-199.9	66.9	38.6	-72.8	24.5	14.2
Total	-524.6	67.8	46.2	-70.5	9.1	6.2

exception is for the techniques using global-level history and P_2 formula, which tend to perform best when using `Threshold` values between 0.2 and 0.4.

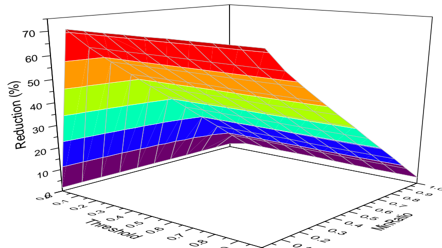
Second, when `Threshold` increases from 0.1 to 1.0, FaMT techniques using P_1 drop more dramatically than techniques using P_2 in terms of reduction. For example, when `Threshold` increases from 0.1 to 0.2, the technique using C_1, method-level history, and P_1 formula drops from 43.3% to 12.52%, while the technique using C_1, method-level history, and P_2 formula does not drop at all. The reason is that using the history of all neighbor mutants unnecessarily lowers the power values of tests and the majority of tests will not have power values of greater than 0.2. The only exceptions are the techniques using statement-level history, which remain stable when `Threshold` increases for both P_1 and P_2. The reason is that if one mutant in a statement is killed, there is a high likelihood that all other mutants in the same statement are also killed, making prioritization using P_1 and using P_2 perform similarly.

Comparison of FaMT with two theoretical techniques. To further investigate FaMT's effectiveness, we further present the reduction of execu-

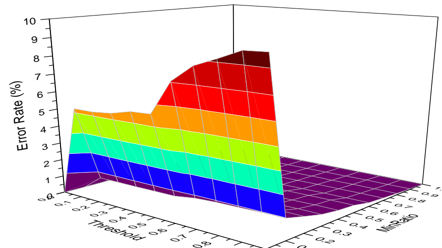
tions by the theoretically worst and best techniques, and compare them with FaMT. The theoretically worst technique executes for each mutant all the tests that cannot kill that mutant before it executes a test that can kill that mutant, while the theoretically best technique executes a test that can kill the mutant first. Table 5.7 presents the reductions achieved by the theoretically worst/best techniques, and an example FaMT technique (i.e., C_3 with the default `Threshold` of 0.3, class-level history, and P_2 formula) over DR. Column 1 lists all the subjects, Columns 2-4 present the reduction of executions for killed mutants, while Columns 5-7 present the reduction of executions for all mutants.

First, the example FaMT technique is close to the theoretically best technique. The theoretically best technique reduces the executions by 14.9% to 90.4% with a total reduction of 67.8%. The example FaMT technique reduces the executions by 5.4% to 67.4% with a total reduction of 46.2%, indicating that FaMT performs closely to the theoretically best technique. For all the subjects, the theoretically worst technique reduces the number of test-mutant executions for killed mutants by -981.8% to -41.8% with a total reduction of -524.6%, i.e., in other words, the theoretically worst technique increases the number of executions by about six times.

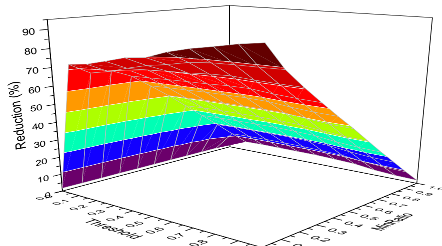
Second, the executions for all mutants cannot be reduced greatly even using the theoretically best prioritization technique. For all the subjects, the theoretically best technique only reduces the number of executions for all mutants by 4.0% to 31.2% with a total reduction of 9.1%. Therefore, FaMT also



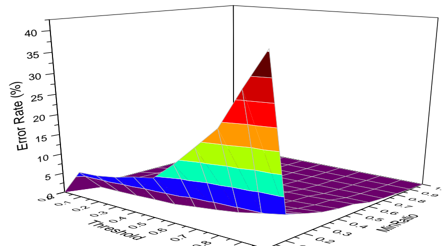
(a) Reduction by FaMT reduction using C_3, statement-level history, and P_1 formula



(b) Error rate by FaMT reduction using C_3, statement-level history, and P_1 formula



(c) Reduction by FaMT reduction using C_3, global-level history, and P_2 formula



(d) Error rate by FaMT reduction using C_3, global-level history, and P_2 formula

Figure 5.3: Reduction and error rate trends on different `Threshold` and `MinRatio` values by FaMT reduction

cannot reduce the number of executions for all mutants greatly: it reduces the number of executions for all mutants by 1.5% to 18.0% with a total reduction of 6.2%. The reason for the low reduction on executions of all mutants is that all prioritization techniques cannot reduce the executions for the mutants that cannot be killed. This finding also further motivates our second study of FaMT reduction.

Table 5.8: Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of all neighbor mutants (P_1)

A.	I.	*TimeMoney*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total	
		Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.
S	*Org*	15.7	0.93	54.5	1.16	33.1	0.42	16.6	0.51	25.4	0.36	20.7	0.51	8.0	0.06	29.9	1.06	30.7	1.09	48.5	0.61
t	C_1	17.8	0.73	53.4	1.76	33.8	0.50	17.8	0.33	24.6	0.55	25.6	0.44	8.2	0.08	30.6	0.45	31.6	0.62	48.1	0.53
a	C_2	15.2	0.98	56.4	0.30	31.6	0.58	13.9	0.69	23.7	0.79	21.2	0.62	7.6	0.05	28.4	1.03	28.1	1.00	49.9	0.60
t	C_3	16.7	0.77	54.9	0.50	34.0	0.52	16.7	0.42	27.3	0.53	27.1	0.50	7.9	0.05	30.7	0.49	30.8	0.53	49.5	0.44
M	*Org*	22.3	0.93	65.6	1.99	48.8	1.68	23.5	1.29	37.4	1.09	34.6	1.12	11.8	0.22	44.5	2.47	47.0	2.55	59.8	1.36
e	C_1	23.4	0.84	64.3	2.76	49.0	1.99	24.3	1.02	38.0	1.57	37.4	0.96	12.0	0.21	44.8	1.76	47.7	2.32	59.1	1.27
t	C_2	21.8	1.02	68.0	0.96	47.9	2.36	22.1	1.49	36.5	1.71	35.5	1.48	11.6	0.24	43.6	2.51	46.1	3.12	61.8	1.48
h	C_3	22.7	0.89	65.9	1.24	49.1	2.10	23.5	1.17	40.2	1.41	39.0	1.08	11.9	0.18	44.8	1.74	48.0	2.28	60.6	1.17
C	*Org*	32.0	1.51	66.9	2.87	54.0	2.08	25.4	1.50	48.4	1.93	44.1	1.58	12.7	0.31	49.4	2.38	51.8	5.27	62.2	1.76
l	C_1	32.5	1.29	65.6	3.75	54.4	1.55	26.5	1.32	47.7	3.52	46.6	1.35	12.9	0.28	49.4	2.30	52.9	4.61	61.4	1.70
a	C_2	32.5	1.65	69.4	1.87	53.5	2.91	24.8	1.59	47.1	2.60	43.9	1.88	12.5	0.27	48.2	2.52	50.3	5.20	64.2	1.84
s	C_3	32.2	1.30	67.2	2.23	53.9	2.56	25.8	1.49	47.3	3.92	47.1	1.45	12.8	0.25	49.4	2.19	52.9	4.39	62.8	1.65
G	*Org*	29.8	1.99	67.2	4.26	56.8	2.94	25.5	1.65	50.9	2.34	52.1	2.65	12.4	0.52	50.2	2.29	54.2	7.47	63.3	2.40
l	C_1	32.6	1.94	65.7	5.60	57.5	2.05	26.7	1.45	51.1	3.65	54.5	2.40	12.6	0.41	50.8	2.33	55.0	6.85	62.4	2.36
o	C_2	30.9	2.37	69.3	3.24	56.6	3.35	24.9	1.78	50.4	2.60	53.5	3.04	12.6	0.49	49.7	2.03	53.0	8.19	65.1	2.46
b	C_3	31.9	2.10	67.3	3.92	57.5	3.18	26.1	1.61	51.2	3.75	54.9	2.59	12.7	0.36	50.6	2.24	55.0	6.57	63.7	2.33

Table 5.9: Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of killed neighbor mutants (P_2)

A.	I.	*TimeMoney*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total	
		Red.	Dev.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.
S	*Org*	10.0	0.21	10.9	0.05	5.5	0.02	6.6	0.08	9.2	0.06	9.8	0.10	1.4	0.00	7.1	0.05	11.9	0.26	10.6	0.07
t	C_1	12.2	0.19	14.1	0.06	6.3	0.02	7.4	0.07	8.1	0.34	14.4	0.13	1.7	0.00	7.3	0.04	11.9	0.23	13.6	0.08
a	C_2	9.3	0.22	4.6	0.10	4.2	0.02	4.0	0.08	8.1	0.29	10.9	0.12	1.1	0.00	5.7	0.04	9.2	0.23	5.3	0.09
t	C_3	11.5	0.20	9.6	0.06	6.5	0.02	6.3	0.08	10.7	0.36	16.2	0.14	1.4	0.00	7.5	0.04	11.2	0.16	10.1	0.09
M	*Org*	13.2	0.33	28.8	0.37	13.4	0.20	12.2	0.45	16.6	0.13	21.8	0.30	2.8	0.05	18.9	0.45	29.8	1.19	27.2	0.34
e	C_1	14.5	0.22	39.2	0.42	13.7	0.13	12.0	0.35	17.6	0.48	24.4	0.35	3.1	0.05	18.1	0.37	26.1	1.14	35.6	0.33
t	C_2	12.7	0.25	19.9	0.51	14.1	0.19	11.2	0.46	16.7	0.44	21.9	0.43	2.7	0.06	19.3	0.46	29.7	1.31	19.7	0.41
h	C_3	14.0	0.20	28.5	0.36	14.3	0.09	11.0	0.40	19.1	0.47	25.8	0.36	3.1	0.05	18.5	0.36	29.5	0.96	27.0	0.33
C	*Org*	18.5	0.76	46.3	0.91	17.5	0.33	14.9	0.52	37.6	0.72	30.5	0.66	4.0	0.08	31.6	0.76	43.9	2.78	42.9	0.63
l	C_1	17.2	0.55	45.8	1.00	18.8	0.22	15.3	0.46	39.3	0.95	33.0	0.60	4.4	0.06	30.8	0.55	41.0	1.93	42.5	0.55
a	C_2	19.7	0.76	46.4	1.17	20.9	0.21	15.2	0.63	39.8	0.99	30.0	0.78	3.9	0.06	31.9	0.96	44.3	2.72	42.7	0.74
s	C_3	17.9	0.61	37.5	0.91	19.0	0.17	14.9	0.51	40.6	0.93	33.9	0.66	4.1	0.05	31.1	0.58	44.5	2.23	35.8	0.58
G	*Org*	16.4	0.88	69.0	2.37	39.4	0.50	15.8	0.62	35.9	1.04	45.7	1.66	4.7	0.04	33.9	1.09	50.7	4.45	63.0	1.20
l	C_1	16.2	0.84	69.1	2.68	43.5	0.50	16.1	0.55	37.7	1.06	46.0	1.52	4.9	0.03	38.9	0.99	49.7	3.67	63.3	1.14
o	C_2	17.7	0.88	69.1	2.03	43.6	0.54	16.0	0.78	42.2	1.24	46.5	1.63	3.9	0.05	33.5	1.23	49.3	4.74	63.3	1.22
b	C_3	16.8	0.82	68.5	2.41	46.2	0.52	16.2	0.62	41.7	1.03	46.2	1.49	4.2	0.03	37.5	1.00	51.3	3.53	62.9	1.12

Table 5.10: Reduction error rates (%) for example FaMT reduction techniques and corresponding random techniques

Tech.	*TimeMoney*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total
	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.
FaMT1	0.93	0.30	1.16	1.27	0.42	0.27	0.51	0.07	0.36	0.30	0.51	0.07	0.06	0.03	1.06	0.28	1.09	0.40	0.61
Rand.	9.73	1.25	11.59	1.32	8.40	1.15	6.54	0.56	18.41	3.23	13.04	0.68	1.38	0.21	11.29	2.31	17.08	1.78	9.58
FaMT2	0.73	0.27	1.76	1.34	0.50	0.21	0.33	0.06	0.55	0.41	0.44	0.05	0.08	0.03	0.45	0.09	0.62	0.27	0.53
Rand.	10.78	1.09	11.56	1.41	8.97	1.13	7.20	0.24	18.35	3.19	14.70	0.49	1.56	0.18	12.60	2.51	17.26	1.84	10.47
FaMT3	0.98	0.29	0.30	0.07	0.58	0.19	0.69	0.07	0.79	0.35	0.62	0.08	0.05	0.02	1.03	0.17	1.00	0.57	0.60
Rand.	9.69	1.05	12.62	0.68	8.03	0.93	6.16	0.21	17.61	2.91	12.98	0.40	1.35	0.15	10.78	2.09	16.38	1.32	9.47
FaMT4	0.77	0.31	0.50	0.60	0.52	0.17	0.42	0.05	0.53	0.40	0.50	0.06	0.05	0.03	0.49	0.09	0.53	0.31	0.44
Rand.	10.42	1.04	11.90	1.35	9.02	1.06	6.80	0.23	19.71	3.51	14.38	0.47	1.53	0.15	12.55	2.55	17.25	1.53	10.31

5.4.5.2 RQ2: FaMT Test Reduction

Evaluation with default threshold and minratio. We applied FaMT reduction techniques with `Threshold` and `MinRatio` both at the default value of 0.3 on all subject programs. Similar with FaMT prioritization techniques, we compared FaMT reduction techniques with *DR* for 20 times for each subject. Table 5.8 presents the mean reduction ratios and mean error rates across 20 runs for all FaMT techniques with P_1 formula. Columns 1 and 2 show the levels of *adaptive* history (denoted as *A.*) and *initial* orderings (denoted as *I.*) used. Columns 3-20 present the reduction ratios and error rates achieved by the studied techniques for each subject. Columns 21 and 22 list the total reduction ratios and error rates over all subjects. Similarly, Table 5.9 presents the experimental results for FaMT reduction using the P_2 formula.

First, all the techniques can reduce the test executions effectively without causing high error rates. In total, when using P_1, the FaMT reduction techniques with both `Threshold` and `MinRatio` of 0.3 reduce test executions by 48.1% to 65.1%, while only causing error rates from 0.44% to 2.46%. When using P_2, the FaMT reduction techniques can reduce the number of test executions by 5.3% to 63.3%, while only causing error rates from 0.07% to 1.22%.

Second, using the P_2 formula gets more conservative reduction than using the P_1 formula. For example, for the statement-level history, techniques using P_2 reduce executions from 5.3% to 13.6% and cause less than 0.1% error rates, while techniques using P_1 reduce executions by higher ratios (from 48.1% to 49.9%) and cause slightly higher error rates (from 0.44% to 0.61%). The

reason is that using P_2 causes tests to have larger power values and tend to stay in the first priority list (i.e., $\mathcal{T}_1$ from Section 5.3.4) and thus be run. One interesting finding is that when the history level becomes global, techniques using P_2 achieve similar execution reductions with techniques using P_1, but cause much lower error rates. For example, a technique using global history and P_2 formula reduces all test executions by 63.3% while causing an error rate of 1.14%, while the corresponding technique using global history and P_1 formula reduces executions by 62.4% while causing a twice as high error rate, 2.36%. The reason is that using the global history of all neighbor mutants unnecessarily lowered the power of tests, causing some *powerful* tests to be moved into the secondary list (i.e., $\mathcal{T}_2$ from Section 5.3.4) and not run later.

Third, there are many techniques that can reduce test executions significantly with negligible error rates. In total, techniques using global-level history and P_2 formula reduce executions by more than 63.0% with less than 1.22% error rates. Techniques using statement-level history and P_1 formula reduce executions by around 50.0% with only around 0.50% error rates. Although the techniques using statement-level history and P_1 formula reduce executions by smaller ratios, their error rates are smaller and more stable. For example, when using the C_3 initial ordering, it only causes error rates of 0.05% to 0.77% across all subjects.

Effectiveness and error rate trends when using various thresholds and minratios. Figure 5.3 illustrates the total reduction and error rate trends for all subjects when two example FaMT reduction techniques use different

`Threshold` and `MinRatio` values from 0.0 to 1.0. The two example techniques are carefully chosen such that they are different enough from each other. The trends for other FaMT reduction techniques should be similar. More precisely, Figures 5.3(a) and 5.3(b) present the reduction and error rate trends for FaMT reduction using the C_3 initial ordering, statement-level history, and P_1 formula. Figures 5.3(c) and 5.3(d) present the reduction and error rate trends for FaMT reduction using the C_3 initial ordering, global-level history, and P_2 formula. In each sub-figure, the two horizontal axes represent `Threshold` and `MinRatio`, and the vertical axis represents the reductions or error rates.

First, when `Threshold` is fixed, if `MinRatio` increases from 0.0 to 1.0, the reductions achieved by FaMT reduction techniques drop linearly for both techniques, while the error rates drop more dramatically when `MinRatio` increases from 0.0 to 0.1. For instance, for the first example technique with `Threshold`=1.0, when `MinRatio` increases from 0.0 to 0.1, the reduction drops from 71.7% to 63 .0%, while the error rate drops from 9.42% to 1.22%. When `MinRatio` increases from 0.1 to 0.2, the reduction ratio drops from 63 .0% to 55.9%, while the error rate only drops from 1.22% to 0.93%. A similar observation can be made for the second example technique. This indicates that using the `MinRatio` of 0.0 is not cost-effective, and using `MinRatio` from 0.5 to 1.0 might cause low reduction. Therefore, using `MinRatio` values between 0.1 and 0.5 can be cost-effective choices.

Second, when `MinRatio` is fixed, if `Threshold` increases from 0.1 to

1.0[2], the reductions achieved by FaMT reduction techniques increase linearly for both techniques, while the error rates increase more dramatically when `Threshold` increases from 0.5 to 1.0. For instance, for the first example technique with `MinRatio` of 0.0, when `Threshold` increases from 0.5 to 1.0, the reduction ratio increases from 71.6% to 71.7%, while the error rate increases from 5.68% to 9.42%. Similarly, for the second example technique with `MinRatio` of 0.0, when `Threshold` increases from 0.5 to 1.0, the reduction ratio increases from 81.3% to 93.4%, while the error rate increases from 9.56% to 40.74%. This indicates that `Threshold` values between 0.1 and 0.5 are more cost-effective than `Threshold` values between 0.5 and 1.0 for FaMT reduction.

Comparison of FaMT reduction with random techniques. To further investigate the effectiveness of FaMT reduction techniques, we compare FaMT techniques with random techniques that execute the same number of tests as FaMT reduction for each mutant. Table 5.10 presents the mean values and standard deviations of error rates caused by the four example FaMT techniques using statement-level history and P_1 formula with corresponding random techniques across 20 runs. Column 1 lists all the compared techniques (each example FaMT technique followed with a random technique). Columns 2-19 list the mean error rates and their standard deviations for each subject. Column 20 lists the overall error rates for all subjects in total.

The error rates caused by random techniques are much larger than those

[2] When `Threshold`=0.0, all the tests are stored in the first priority list and thus executed, and the reduction ratios are close to 0.

Table 5.11: Comparison between FaMT and regression test prioritization and reduction

Sub	Test Prioritization (%)			Test Reduction (%)			
	FaMT	Tot.	Add.	FaMT		Greedy	
TimeMoney	17.0	-15.7	4.7	16.7	(0.77)	30.1	(3.99)
Jaxen	67.4	-144.0	55.8	54.9	(0.50)	72.7	(2.62)
Xml-Sec	28.8	-29.4	-9.2	34.0	(0.52)	38.2	(4.12)
Com-Lang	8.8	-12.5	5.2	16.7	(0.42)	24.3	(1.97)
JDepend	29.4	-4.8	18.3	27.3	(0.53)	45.2	(3.15)
Joda-Time	38.3	5.5	27.0	27.1	(0.50)	50.5	(3.97)
JMeter	5.4	-2.7	3.3	7.9	(0.05)	5.2	(0.09)
Mime4J	23.9	-30.7	-1.0	30.7	(0.49)	31.8	(1.61)
Barbecue	38.6	-46.5	-18.0	30.8	(0.53)	53.4	(16.59)

of FaMT techniques although they reduce the executions to the same extent. For example, the first FaMT technique causes an error rate of 0.61% in total for all subjects, while the corresponding random technique causes an error rate of 9.58%. In addition, the error rates caused by FaMT techniques are more stable than those of random techniques. For example, the standard deviations of error rates caused by the first FaMT technique range from 0.03% to 1.27%, while the standard deviations of error rates caused by the corresponding random technique range from 0.21% to 3.23%.

5.4.5.3 RQ3: Comparison with Regression Techniques

Table 5.11 summarizes the comparison of two example FaMT techniques and traditional regression testing techniques. Column 1 lists all the subjects. Columns 2-4 present the mean reduction of executions for killed mutants (across 20 runs for each subject) achieved by the example FaMT prioritization technique (using the C_3 initial ordering, class-level history, P_2 formula, and default `Threshold`) with the *total* and *additional* regression test

prioritization techniques. Columns 5 and 6 present the mean reduction of executions for all mutants with mean error rates in brackets (across 20 runs) achieved by the example FaMT reduction technique (using C_3 initial ordering, statement-level history, P_1 formula, and default `Threshold` and `MinRatio`) and the *greedy* regression reduction technique.

Both regression prioritization techniques do not always reduce the number of executions for killed mutants. For example, the *total* technique reduces executions by -144.0% to 5.5%, while the *additional* technique reduces executions by -18.0% to 55.8%. In contrast, the example FaMT prioritization effectively reduces executions from 5.4% to 67.4% for all subjects. We believe the reason is that regression prioritization techniques were not originally designed for mutation testing. One interesting finding is that the *additional* regression prioritization technique is able to reduce executions effectively for several subjects. For example, it reduces executions by more than 10% for three subjects. The reason is that diverse tests tend to be executed early against each mutant because the *additional* technique always picks the test that covers most uncovered program elements as the next test. The early execution of diverse tests may have a higher probability to kill a mutant earlier.

The *greedy* regression reduction technique can significantly reduce the number of executions for all subjects (from 5.2% to 72.7%). However, the error rates caused by it can be extremely high for some subjects, e.g., 16.59% for *Barbecue*, making it not suitable for reducing the cost of mutation testing. In contrast, although the example FaMT reduction reduces executions

Table 5.12: Runtime overhead by FaMT techniques

Sub	Javalanche Time (s)	FaMT Prioritization P_1 (s)	FaMT Prioritization P_2 (s)	FaMT Reduction P_1 (s)	FaMT Reduction P_2 (s)
TimeMoney	433	0.04	0.04	0.03	0.04
Jaxen	2901	18.34	17.68	17.72	17.47
Xml-Sec	3184	0.91	0.70	0.89	0.59
Com-Lang	4475	0.71	0.70	0.70	0.69
JDepend	182	0.07	0.06	0.05	0.06
Joda-Time	11788	8.55	8.13	8.16	8.28
JMeter	3452	0.22	0.16	0.15	0.13
Mime4J	10880	0.41	0.44	0.47	0.41
Barbecue	455	0.10	0.08	0.09	0.09

by smaller ratios (from 7.9% to 54.9%), it incurs small and stable error rates (from 0.05% to 0.77%), demonstrating that FaMT reduction is more suitable than regression reduction for reducing mutation testing cost.

5.4.5.4 RQ4: FaMT Efficiency

We measured the runtime overheads for all FaMT prioritization and reduction techniques. Due to the space limitation, we only show the overheads for the most expensive techniques that use C_3 (i.e., the heuristic that combines C_1 and C_2, thus needing more time to calculate) and the global history. The runtime overheads for other FaMT techniques are no more than the ones shown. In Table 5.12, Column 1 lists the subjects, and Column 2 lists the total execution time for the state-of-the-art Javalanche tool to calculate the mutation score. Columns 3 and 4 list the execution time for the example FaMT prioritization techniques using P_1 and P_2, respectively. Similarly, Columns 5 and 6 list the execution time for the reduction techniques using P_1 and P_2, respectively. All the presented four techniques cause similar overheads. The reason is that all the techniques are based on the same basic algorithm (Algo-

rithm 3). The overhead for each technique varies a lot across different subjects, because FaMT techniques cost more for subjects with a larger number of mutants or larger sets of tests that reach each mutant. The results also show that FaMT costs at most 18.34s (on *Jaxen*), which is negligible compared to the cost of mutation testing (2901s on *Jaxen*) and demonstrates the efficiency of FaMT.

5.4.5.5 Threats to Validity

Threats to external validity. First, although we used 9 medium-sized Java programs, our results may not be generalizable to other programs. Second, the results may not be generalizable to other test suites. Third, our results based on mutants generated by Javalanche may not be generalizable to mutants generated by other tools.

Threats to internal validity. The main threat to internal validity for our study is that there may be faults in our implementation of the 352 variant prioritization techniques and 3872 variant reduction techniques of FaMT, as well as the other controlled techniques. To reduce this threat, we reviewed all the code that we produced for our experiments before conducting the experiments.

Threats to construct validity. The main threat to construct validity is the metrics that we used to assess the effectiveness and cost of our techniques. To reduce this threat, we used the ratio of executions reduced to assess the techniques' effectiveness, and used the runtime overhead to assess

the techniques' cost. We also used error rate to measure the approximation caused by FaMT reduction.

5.5 Summary

This chapter makes the following contributions:

- **Idea:** We introduce the general idea of optimizing mutation testing using test prioritization and reduction. The cost of mutation testing has two key elements—executing some tests for killed mutants and executing every test for non-killed mutants—and test prioritization and reduction address both of these elements.

- **Techniques:** We embody our idea in algorithms for test prioritization and reduction, and we define a family of techniques. These techniques are based on coverage information about test execution (e.g., the number of times the test reaches the mutated statement while executing on the original, unmutated program), on the history of test executions on other mutants (e.g., the accumulating number of mutants that the test killed or did not kill before executing the current mutant), and on the history granularity level (e.g., history at the statement level or the method level).

- **Evaluation:** We plan evaluate all 352 variant techniques of FaMT prioritization and 3872 variant techniques of FaMT reduction on real-world Java programs to show the effectiveness and efficiency of FaMT.

Chapter 6

Mutation Testing for Fault Localization in Regression Testing

The pervious two chapters (Chapters 4 and 5) presented two unification approaches, each using regression testing techniques to help with mutation testing. This chapter presents an approach that uses mutation testing to address the fault localization problem in the context of regression testing. This chapter is based on our paper presented at the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (SPLASH/OOPSLA 2013) [158].

6.1 Introduction

Software systems usually undergo evolution to add new features, fix bugs, and refactor existing code. It is hard to keep software evolution free from faults. Therefore, regression testing has been utilized to validate the program edits during software evolution. When regression tests fail after edits, developers may need to localize and fix the failure-inducing program edits.

The problem of *fault localization*, i.e., identifying the locations of faulty lines of code, remains challenging, often requiring much manual effort. This

chapter presents a novel solution to this problem in the context of code that evolves. Our insight is that the essence of failure-inducing edits made by the developer can be simulated using mechanical program transformations. Specifically, some transformations are likely to share the same locations with failure-inducing edits if those transformations transform the old program version (i.e., the version right before the faults were introduced) to have similar test pass/fail results as the new version with real developer edits.

To simulate developer edits, we use program transformations based on *mutation testing* [20,32,42,46,49,53,96,121,122,156], which is a methodology originally designed for measuring test-suite quality based on injected faults. Mutation testing generates variants (termed *mutants*) for the original program under test using mechanical transformation rules (termed *mutation operators*). Each mutant is the same with the original program except for the mutated statement. A mutant is termed *killed* by a test suite if some test from the suite produces different results on the mutant and the original program. Empirical studies have shown that test suites that kill a high percentage of mutants are likely to reveal more faults and mutation testing is often viewed as the strongest test criteria [12,41]. It has been used to evaluate the quality of existing test suites [32,96,122,131] and to generate test suites with high quality [34,42,49, 95,156].

To our knowledge, mutation changes have not been utilized to simulate developer edits to achieve precise fault localization. The existing approaches for fault localization during software evolution mainly use sole coverage in-

formation of developer edits. *Change impact analysis* [110] is a well-known methodology for determining *affecting changes*, i.e., a subset of program edits that might have caused the test failure, based on edit coverage information in regression testing [51, 110, 150]. It has been shown that the number of affecting changes for each failed test can still be large [150]. Therefore, various techniques have been proposed to localize failure-inducing changes more precisely [109, 128, 150]. The recently developed FaultTracer approach [150] introduces spectrum-based fault localization [8, 66, 77, 144] to localization of failure-inducing edits; experimental results show that FaultTracer significantly outperforms Ren et al.'s ranking heuristic [109] based on test call graph structures. However, FaultTracer still suffers from lack of accuracy, because the *spectrum* information (i.e., the edits that are mainly executed by failed tests are considered more suspicious) is still based on only coverage information and the suspicious edits may not be responsible for test failures. For instance, some edits are ranked at the top just because they are *accidentally* executed by failed tests.

A straightforward idea to refine the fault localization results is to automatically apply various subsets of program edits according to their ranking to localize failure-inducing edits more precisely. However, there are three basic reasons that make this idea impractical. First, program edits might have complex compilation dependences between them, which does not allow them to be applied independently. Second, iteratively applying various combination of edits may cost extra test execution time. Third, iteratively applying

program edits does not work for concurrent programs, since some faulty edits may be missed just because they accidentally passed the test suite once. As a result, existing techniques for localizing failure-inducing edits usually recommend *manually* applying and inspecting edits after ranking them [23,109,150].

Modern software systems usually undergo evolution to add new features, fix bugs, or refactor existing code. The extensive use of subtyping and dynamic dispatch in object-oriented programming makes it difficult to analyze root causes (i.e., failure-inducing edits) for software failures during software evolution. Therefore, a large body of research has been dedicated to localizing failure-inducing program edits for object-oriented languages [23, 109, 110, 128, 150].

Change impact analysis [110] is a well-known methodology for determining affecting changes, i.e., a subset of program edits that might have caused the test failure, for each failed test. It has been shown that the number of affecting changes for each failed test can still be large in number [150]. Therefore, various techniques have been proposed to localize failure-inducing changes more precisely [109, 128, 150]. As a state-of-the-art approach, FaultTracer [150] introduces the spectrum-based fault localization methodology [8, 66, 77, 144] to localize failure-inducing edits, and has been shown to outperform Ren et al.'s ranking heuristic [109] based on test call graph structures significantly. However, FaultTracer still has the accuracy problem, because it only uses the *spectrum* information (i.e., the edits that are mainly executed by failed tests are more suspicious), and the suspicious edits may have no *impact* to the test

execution. For instance, some edits are ranked top might just because they are accidentally executed by failed tests.

Mutation testing [20, 32, 42, 46, 49, 53, 96, 121, 122, 156] is a fault-based methodology for enabling testing programs with high-quality test suites. Generally speaking, mutation testing generates variants (known as *mutants*) for the original program under test based on mechanical transformations (known as *mutation operators*). Each mutant is the same as the original program except the mutated statement. A mutant is denoted as *killed* by a test suite if some tests of the suite derive different results on the mutant and the original program. Since each mutant can be treated as a faulty version of the original program, a test suite that can kill more mutants has the potential to reveal more real faults. Therefore, mutation testing has been used to evaluate the quality of existing test suites [32, 96, 122, 131], and to generate test suites with high quality [34, 42, 49, 95, 156].

Although real program edits by developers and mechanical mutation changes by mutation testing are both changes to the original program, they are traditionally treated as two separate dimensions. This chapter unifies these two dimensions of changes. We use both the spectra of edits (obtained using FaultTracer) as well as the potential impacts of edits (simulated by mutation changes) to achieve more accurate fault localization.

This chapter presents our methodology of fault injection for fault localization (FIFL) and our framework that embodies it for achieving more precise fault localization during software evolution based on mutation testing. To lo-

calize failure-inducing edits, FIFL first utilizes the mutation testing results on the old version[1] and gets the test execution results for each mutant. Second, following FaultTracer, FIFL uses spectrum-based techniques [8, 66, 77, 144] to calculate the suspiciousness of program edits between the old and new versions. Third, FIFL builds a mapping between program edits with mutants of the old version that can potentially simulate the corresponding edits based on a set of inference rules. Fourth, FIFL determines the impacts of mutation changes by calculating the similarity between test execution results (Pass/Fail) of the mutants for the old version with the test execution results of the new version, and treats the similarity as the suspiciousness of mutants. Finally, for every program edit, FIFL refines its suspiciousness based on the suspiciousness of its mapped mutants, because those mutants can simulate the potential impact of the edit.

We believe our basic insight into unifying mutation changes with developer edits is also applicable to other key software testing realms. For example, mutation testing results for the old program version can optimize test selection [51] and prioritization [116] for the new version, because the potential impact of program edits can be simulated by existing mutants. We plan to establish these connections in future work.

```
1  public class BankAcnt{
2    public static String bank="ABank"
3    public String account;
4    public double saving=100;
5    public double iRate=0.01;
6    public double iRate2=0.02;
7    public Acnt(String a){account=a;}
8    public double getBalance()
9    {return saving;}
10   public double withdraw (double v) {
11     if(saving>=v) {
12       saving = saving-v;
13       return v;
14     }else return 0;
15   }
16   public void deposit (double v)
17   {saving = saving+v;}
18   double getRate(){return iRate;}
19   double getRate2(){return iRate2;}
20 }
21 public class SuperAcnt extends
       BankAcnt {
22   public double iRate=0.03;
23   public SuperAcnt(String a){super(a)
       ;}
24   public void deposit(double v) {
25     //fault, "0" should be "v"
26      saving=saving+0;
27     if(v>=50){saving=saving+1.0;}
28   }
29 }
30          (a)
```

```
1  public class TestSuite{
2    void test1() {
3      BankAcnt acnt=new BankAcnt("acnt1
           ");
4      acnt.withdraw(20);
5      double rate=acnt.getRate();
6      assertEquals(acnt.getBalance(),
           80);
7    }
8    void test2() {
9      SuperAcnt acnt=new SuperAcnt("
           acnt1");
10     acnt.withdraw(40);
11     assertEquals(acnt.getBalance(),
           60);
12   }
13   void test3() {
14     SuperAcnt acnt=new SuperAcnt("
           acnt1");
15     acnt.deposit(0);
16     assertEquals(acnt.getBalance(),
           100);
17   }
18   void test4() {
19     BankAcnt acnt1=new BankAcnt("
           acnt1");
20     SuperAcnt acnt2=new SuperAcnt("
           acnt2");
21     double amount=acnt1.withdraw(80);
22     double rate1=acnt1.getRate();
23     acnt2.deposit(amount);
24     double rate2=acnt2.getRate();
25     assertEquals(acnt2.getBalance(),
           180);
26   }
27 }
28          (b)
```

Figure 6.1: (a) Example in evolution. Note that methods/fields in box are added, methods/fields with line-through are deleted, and methods/fields with underlines are changed. The statements with underlines inside changed methods are added. (b) Tests for the example.

6.2 Example

In this section, we use the example in Figure 6.1 to illustrate the FaultTracer approach for localizing failure-inducing edits and to motivate our FIFL approach. Figure 6.1 (a) shows the edited program, which manages the basic bank account functionality of two account types, i.e., `BankAcnt` (basic account type), `SuperAcnt` (super account type). Figure 6.1 (b) presents the regression test suite for validating the edits made on the example program. Assuming that the developer made a failure-inducing edit when adding `SuperAcnt.deposit()`[2] (shown in gray), `test4` then fails and detects the fault. The goal is to identify the failure-inducing edit precisely. We first show the steps applied by FaultTracer, then we show the limitations of FaultTracer and the intuition of FIFL.

Following traditional change impact analysis [110,150], FaultTracer first extracts the edits between program versions as *atomic changes*, denoted as Δ. Atomic changes are categorized as added methods (`AM`), deleted methods (`DM`), changed methods (`CM`), added fields (`AF`), deleted fields (`DF`), changed instance fields (`CFI`), changed static fields (`CSFI`), field lookup changes ($\texttt{LC}_f$) due to the field hiding hierarchy changes, and method lookup (i.e., dynamic dispatch) changes ($\texttt{LC}_m$) due to the method overriding hierarchy changes. Note that FaultTracer splits all higher-level changes (e.g., class changes) into atomic

[1] For evolving software systems, the mutation testing results for the old version may be already available in the repository, and ready to use.

[2] Please note that in this chapter we omit the parameters and return types for methods to make the method names shorter.

changes. FaultTracer also infers dependences between atomic changes. For example, a method/field lookup change is dependent on the method/field addition or deletion that causes the lookup change. A non-lookup change c_1 (e.g., `CM` or `AM`) is dependent on another atomic change c_2 iff applying c_1 to the original program version without applying c_2 results in a syntactically invalid program. FaultTracer extracts atomic changes `AM(SuperAcnt.deposit())`, $\texttt{LC}_m$`(SuperAcnt, SuperAcnt.deposit())`[3], `DF(BankAcnt.iRate2)`, etc., for Figure 6.1. For the change dependences, `DF(BankAcnt.iRate2)` is determined to be dependent on `DM(BankAcnt.getRate2())` (`DM(BankAcnt.getRate2())` $\preceq$ `DF(BankAcnt.iRate2)`), as deleting `BankAcnt.iRate2` without deleting method `BankAcnt.getRate2()` can cause `BankAcnt.getRate2()` to access a field without definition. FaultTracer also infers $\texttt{LC}_m$`(SuperAcnt, SuperAcnt.deposit())` depends on addition `AM(SuperAcnt.deposit())` (`AM(SuperAcnt.deposit())` $\preceq$ $\texttt{LC}_m$`(SuperAcnt, SuperAcnt.deposit())`), since the `AM` change causes the $\texttt{LC}_m$ change.

Second, FaultTracer determines the set of *affected tests* in the regression suites that have been influenced by the program edits based on the precise Extended Call Graph (ECG) analysis [150]. For each affected test, FaultTracer further identifies the set of atomic changes that might have changed the test's behavior, and denotes them as *affecting changes* for the test. For the example program, all the four tests are affected tests, and their affecting changes are

[3] An $\texttt{LC}_m$ change $\texttt{LC}_m$`(R,S.m())` models the fact that an invocation to method `S.m()` on an object with run-time type `R` results in a different target method due to method additions or deletions during evolution.

Table 6.1: Suspiciousness Calculation for Developer Edits and Mutation Changes.

Edits	Mapping Mutants	Affected Tests				Suspiciousness Score			
		test1	test2	test3	test4	Tarantula	SBI	Jaccard	Ochiai
CFI(BankAcnt.iRate)		✓			✓	0.75	0.50	0.50	0.71
CM(BankAcnt.withdraw())		✓	✓		✓	0.60	0.33	0.33	0.58
AF(SuperAcnt.iRate)					✓	1.00	1.00	1.00	1.00
AM(SuperAcnt.deposit())				✓	✓	0.75	0.50	0.50	0.71
CFI(BankAcnt.iRate)	line 5: BankAcnt.iRate=1.0					0.00	0.00	0.00	0.00
	line 5: BankAcnt.iRate=0.0					0.00	0.00	0.00	0.00
	line 5: BankAcnt.iRate=-1.0					0.00	0.00	0.00	0.00
CM(BankAcnt.withdraw())	line 12: saving = saving+v;	✓	✓			0.00	0.00	0.00	0.00
	line 12: saving = saving/v;	✓	✓			0.00	0.00	0.00	0.00
	line 12: saving = saving*v;	✓	✓			0.00	0.00	0.00	0.00
	line 12: saving = saving%v;	✓	✓			0.00	0.00	0.00	0.00
AF(SuperAcnt.iRate)	line 5: BankAcnt.iRate=1.0					0.00	0.00	0.00	0.00
	line 5: BankAcnt.iRate=0.0					0.00	0.00	0.00	0.00
	line 5: BankAcnt.iRate=-1.0					0.00	0.00	0.00	0.00
AM(SuperAcnt.deposit())	line 17: saving = saving-v;				✓	1.00	1.00	1.00	1.00
	line 17: saving = saving/v;			✓	✓	0.75	0.50	0.50	0.71
	line 17: saving = saving*v;			✓	✓	0.75	0.50	0.50	0.71
	line 17: saving = saving%v;			✓	✓	0.75	0.50	0.50	0.71
Out		P	P	P	F				

shown in Columns 3-6 in the upper part of Table 6.1. A checkmark denotes that an atomic change is an affecting change of a affected test.

Third, FaultTracer uses the correlation between tests and affecting changes to determine the potential failure-inducing edits. The basic intuition is that an affecting change that is mainly executed by failed tests rather than passed tests is more suspicious. Therefore, FaultTracer utilizes the correlation between affecting changes and the failed tests to calculate the suspiciousness score for each affecting change. Columns 7-10 of Table 6.1 show the suspiciousness score for each affecting change calculated by four well-known suspicious calculation formulae, i.e., Tarantula [66], Statistical Bug Isolation (SBI) [77],

Jaccard [8], and Ochiai [8, 144]. However, the localization results are not ideal for this example: all the four formulas rank the real failure-inducing edit `AM(SuperAcnt.deposit())` as the tied third suspicious edit. The reason is that `AM(SuperAcnt.deposit())` is executed by `test3`, which passed, and `AF(SuperAcnt.iRate)` happens to be executed by the only failed test, thus mistakenly lowering the rank of `AM(SuperAcnt.deposit())` and lifting the rank of `AF(SuperAcnt.iRate)`.

The basic intuition of FIFL is that the mutation changes made by mutants of the old program version are able to simulate the impacts of developers' edits, and make the test execution results on some suspicious mutants (which share the same locations with real failure-inducing edits) similar to the test execution results on the new program version (with program edits). Therefore, we can directly use the existing mutation testing results of the old program version to boost the fault localization results while avoiding the drawbacks of iteratively applying subsets of program edits. For example, we can use the mutants occurring on the statements inside the same code element with `CFI(BankAcnt.iRate)` or `CM(BankAcnt.withdraw())` to simulate the effect of these two edits. For `AM(SuperAcnt.deposit())` and `AF(SuperAcnt.iRate)`, we cannot find the statements that share the same code elements with them because they do not exist on the old version. After analyzing the program, we find that a mutant occurring in `BankAcnt.deposit()` has a similar impact with adding `SuperAcnt.deposit()`, because the invocation to `SuperAcnt.deposit()` was directed to `BankAcnt.deposit()` in the

old version. Therefore, adding `SuperAcnt.deposit()` may have a similar impact with changing `BankAcnt.deposit()` (using mutation). Similarly, we find that a mutant occurring in `BankAcnt.iRate` has a similar effect with editing a `SuperAcnt.iRate`, since `SuperAcnt.iRate` hides `BankAcnt.iRate` in the new version (detailed change mapping inference is shown in Section 6.3.1). For each edit, Column 2 of the lower part of Table 6.1 shows some example mapping mutants that may simulate each edit. Columns 3-6 show the test execution results for each mutant of the old program version. A checkmark denotes a mutant is killed by a test, i.e., the test fails on the mutant. Intuitively, we can find that any mapping mutants of the first three edits cannot fail the real failed test, `test4`, while a mapping mutant (in `BankAcnt.deposit()`) of the real failure-inducing edit, `AM(SuperAcnt.deposit())`, has exactly the same test execution results with the test execution results after evolution, indicating the benefits of improving edit suspiciousness calculation based on mutation testing. The detailed fault localization for this example will be further illustrated in Section 6.3.3.

6.3 Approach

Figure 6.2 shows the general framework of FIFL. Assume we have two program versions during software evolution, P and P', together with their regression test suite T, which passes on P but failed on P'. First, traditional mutation testing process is applied on P: generating all the mutants M for P and recording the mutant execution results, i.e., the correlation between

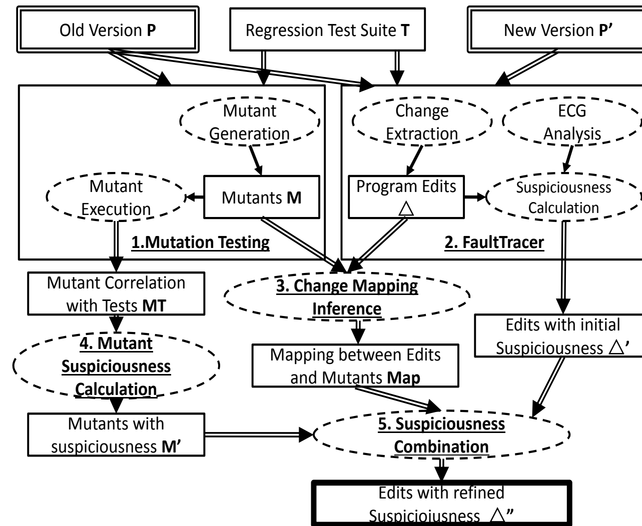


Figure 6.2: FIFL architecture.

mutants and the tests that kill them (denoted as MT). As FIFL only requires mutation testing results on the old version, FIFL recommends that this step is performed in the background in parallel with developing the new version, and thus the mutation testing results are directly available before applying FIFL. Second, FIFL extracts edits between P and P' and calculates the suspiciousness of each edit using FaultTracer [150], which utilizes the spectrum information of edits and assigns a higher suspiciousness value to a edit if it is mainly executed by failed tests. Third, FIFL infers the mapping between developer edits and mutants based on a set of inference rules. Fourth, FIFL calculates the suspiciousness value for each mutant. A mutant is assigned with a higher suspiciousness value if it has a similar impact to regression tests with the program after edits, P'. Finally, FIFL refines the suspiciousness values

$$\frac{c \in CM \cup DM \quad \mu \in M \quad s_{\mu} \sqsubset_P m_c}{c \mapsto \mu} (1)$$

$$\frac{c \in CFI \cup CSFI \cup DF \quad \mu \in M \quad s_{\mu} \sqsubset_P f_c}{c \mapsto \mu} (2)$$

$$\frac{c \in AM \quad \mu \in M \quad s_{\mu} \sqsubset_P m \quad m_c \rightarrow_{P'} m}{c \mapsto \mu} (3) \qquad \frac{c, c' \in AM \quad \mu \in M \quad c' \mapsto \mu \quad m_c \rightarrow_{P'} m_{c'}}{c \mapsto \mu} (4)$$

$$\frac{c \in AF \quad \mu \in M \quad s_{\mu} \sqsubset_P f \quad f_c \dashrightarrow_{P'} f}{c \mapsto \mu} (5) \qquad \frac{c, c' \in AF \quad \mu \in M \quad c' \mapsto \mu \quad f_c \dashrightarrow_{P'} f_{c'}}{c \mapsto \mu} (6)$$

$$\frac{c \in AM \quad c' \in DM \quad c'' \in LC_m \quad \mu \in M \quad s_{\mu} \sqsubset_P m_{c'} \quad c \preceq c'' \quad c' \preceq c''}{c \mapsto \mu} (7)$$

$$\frac{c \in AF \quad c' \in DF \quad c'' \in LC_f \quad \mu \in M \quad s_{\mu} \sqsubset_P f_{c'} \quad c \preceq c'' \quad c' \preceq c''}{c \mapsto \mu} (8)$$

$$\frac{c \in AM \cup AF \quad c' \in \Delta \quad \mu \in M \quad c' \mapsto \mu \quad c \preceq c'}{c \mapsto \mu} (9)$$

Figure 6.3: Rules for inferring change mapping.

of program edits (based on spectrum information) using the suspiciousness of their mapping mutants (based on impact information), and returns the edits with final suspiciousness values, Δ'', as the final result. As the first two steps (i.e., mutation testing and FaultTracer) are based on existing techniques, the following subsections show the change mapping inference (Section 6.3.1), mutant suspicious calculation (Section 6.3.2), and suspiciousness combination (Section 6.3.3) in detail.

6.3.1 Change Mapping Inference

In order to bridge the developer edits between P and P' with the mechanical changes to P via mutation, FIFL defines a set of inference rules for inferring the mapping between them. Figure 6.3 shows the inference rules. In the figure, m_c denotes the corresponding method for method-level change c, f_c denotes the corresponding field for field-level change c, and s_{μ} denotes the mu-

```
    public StringBuffer format(Calendar calendar, StringBuffer buf) {
866:    if (mTimeZoneForced) {
867:        calendar.getTimeInMillis();
868:        calendar = (Calendar) calendar.clone();
869:        calendar.setTimeZone(mTimeZone);
870:    }
871:    return applyRules(calendar, buf);
    }
```

Figure 6.4: Code snippet of *Com-Lang* V3.03 to illustrate change mapping for `CM` edits

tated statement of a mutant μ. $s \sqsubset_P x$ denotes that statement s is within the scope of method or field x in the old program version P. $f \dashrightarrow_{P'} f'$ denotes that field f hides field f' in the new version P', and $m \rightarrow_{P'} m'$ denotes that method m overrides method m' in P'. $c \preceq c'$ denotes change c' depends on change c. Finally, $c \mapsto \mu$ denotes that edit c is mapped with mutant μ. To better motivate and illustrate the rules, we present simple artificial examples as well as real-world code snippets to show how the change mapping can help increase the suspiciousness of failure-inducing edits and/or decrease the suspiciousness of fault-free edits.

6.3.1.1 Inference for Changed/Deleted Elements

Shown by the first two rules, for modifications and deletions of methods and fields, the mapping is trivial: FIFL just maps a change with a mutant if the change and the mutant occur in the same method or field (since the method or field exists in the old version). The mapped developer edits and mutation changes occur at the same functional point and thus these two dimensions of changes may have similar impacts to the program under test.

For example, when *Com-Lang* evolves from V3.02 to V3.03, the developer changed the `format()` method of class `FastDateFormat` and removed lines 866 to 870. As the changed method is executed by 35 tests with only one failed, making traditional approaches based on spectrum information not able to localize the fault precisely. In contrast, FIFL directly maps the `CM` change to the 5 mutants occurred inside the `format()` method in V3.02. Within the mapped mutants, 3 mutants, which remove method invocations for line 867 to line 869 respectively, have exactly the same failed test as V3.03, demonstrating that the mapped mutants can be used to simulate the effect of real method changes. This mapping significantly increase the ranking of the failure-inducing edit (details shown in Section 6.5).

6.3.1.2 Inference for Added Elements Overridding/Hiding Existing Elements

The mutant mapping inference rules for additions of fields and methods are more complex because they have no corresponding code elements in the old version. We illustrate those rules with examples in Figure 6.6. In the figure, we connect an added element (`AM` or `AF`) with another non-added element using a twin line if and only if the added element can be mapped to the mutant within the scope of the non-added element. The basic intuition for Rules 3-6 is that the mutants occurring in a method/field c' that is close to the added method/field c in the overriding/hiding hierarchy (such that an invocation/access to c actually executes c' in the old version) can be used to simulate the impact of adding c, because both adding c and mutating c' may

change the execution of c'. Rules 3 and 4 define the mapping inference for method additions that override some methods: If the added method overrides some existing methods that are not newly added, Rule 3 maps the addition change to all mutants that occur inside the existing method; if the atomic change of the method overridden by the added method is already mapped with a set of mutants, Rule 4 also maps the added method with those mutants. Similarly, Rules 5 and 6 infer the change mapping for field additions that hide other fields: If the added field hides some existing fields that are not newly added, Rule 5 maps the addition change to all mutants that occur inside the existing field; otherwise, if the change on the field hidden by the added field is already mapped with a set of mutants, Rule 6 also maps the added field to those mutants. Note that Rules 4 and 6 should be iteratively applied until they reach a fix point. To illustrate, the change mapping inference steps for Figure 6.6(a) are shown as follows:

Applied Rules	AM(m_3)	AM(m_4)
Rule 3	M_{m_2}	-
Rule 4	M_{m_2}	M_{m_2}

where M_{m_2} denotes the mutants occurring in the body of method m_2. Similarly, the two `AF` changes in Figure 6.6(b) are mapped with mutants inside f_1 and f_3.

For example, when *Joda-Time* evolves from V1.10 to V1.20, the fault-free edit `AM(getDateTimeMillis())` in class `BasicChronology` was ranked high because it was executed by some failed tests accidentally. As the method was newly added, Rule 1 cannot map the edit with any mutants of the old

```
class org.joda.time.chrono.AssembledChronology:
public long getDateTimeMillis(int year,
    int monthOfYear, int dayOfMonth, int hourOfDay,
    int minuteOfHour, int secondOfMinute, int millisOfSecond) {...}
►class org.joda.time.chrono.BasicChronology:
  public long getDateTimeMillis(int year,
      int monthOfYear, int dayOfMonth, int hourOfDay,
      int minuteOfHour, int secondOfMinute, int millisOfSecond) {...}
```

Figure 6.5: Code snippet of *Joda-Time* V1.20 to illustrate change mapping for `AM` edits with overridden methods

version. Figure 6.5 shows the class inheritance hierarchy for the class containing the added method. In the figure, ► denotes the below class is a subclass of the above class. As the added method overrides an existing method in class `AssembledChronology` (denoting that the two methods have similar functionalities), the invocation to the specific functionality of the new method may be resolved to the overridden method in the old version. Thus, some mutation changes to the old overridden method may have similar impacts with the edit of adding the faulty method, and thus the mutants in the overridden method can be mapped to the `AM` edit to simulate its impact. In fact, using this mapping, FIFL successfully decreases the suspiciousness of the fault-free edit.

6.3.1.3 Inference for Added Elements Sharing Overriding/Hiding Hierarchy with Deleted Elements

There may also be some added method/field c that shares the same overriding/hiding hierarchy with some deleted method/field c', i.e., although they never co-exist in one version, they may implement the same functional-

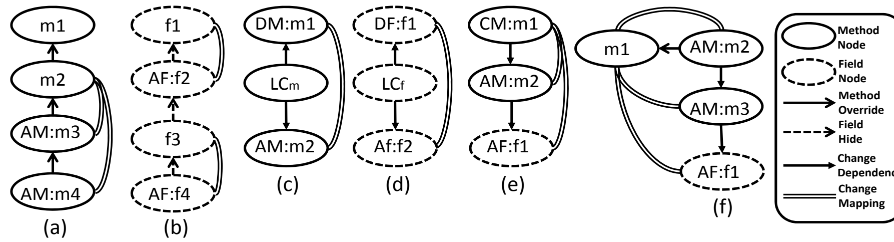


Figure 6.6: Illustration for mutant mapping.

ity. Therefore, the mutants within $m_{c'}$ in the old version may also be able to simulate the impact of c. As added and deleted elements do not exist in the same version, FIFL cannot use the ordinary overriding/hiding hierarchy analysis to infer the change mapping. Instead, FIFL utilizes the fact that both addition changes and deletion changes would cause method or field lookup changes, and uses those lookup changes to bridge the mapping between addition changes and mutants in deleted elements. For any method/field addition c, if some method/field deletion c' causes the same method/field lookup change with c, Rule 7/8 maps the mutants inside the corresponding deleted element of c' with c. Figures 6.6(c) and 6.6(d) illustrate that the mutants within the deleted methods/fields can be mapped with addition changes.

In the same revision with the last example (i.e., when *Joda-Time* evolves from V1.10 to V1.20), the fault-free edit `AM(getAverageMillisPerMonth())` in class `BasicFixedMonthChronology` was ranked high by the existing Fault-Tracer approach, because it was executed by the failed tests accidentally. As the added method does not override any existing method, Rules 3 and 4 cannot map the edit with any mutant. Figure 6.7(b) shows the inheritance hierarchy

```
class org.joda.time.chrono.AssembledChronology
►class org.joda.time.chrono.BaseGJChronology
  ►class org.joda.time.chrono.CopticChronology:
    long getAverageMillisPerMonth() {...}
```

(a) *Joda-Time* V1.10

```
class org.joda.time.chrono.AssembledChronology
►class org.joda.time.chrono.BasicChronology
  ►class org.joda.time.chrono.BasicFixedMonthChronology:
    long getAverageMillisPerMonth() {...}
    ►class org.joda.time.chrono.CopticChronology
```

(b) *Joda-Time* V1.20

Figure 6.7: Code snippets of *Joda-Time* V1.10 and V1.20 to illustrate change mapping of `AM` edits with deleted methods sharing the same overriding hierarchy

for the class of the added method (`BasicFixedMonthChronology`). The containing class of the edit has a superclass named `AssembledChronology` and a subclass named `CopticChronology`. Shown in Figure 6.7(a), the old version has a method (which was deleted in the new version) with the same signature and under the same class inheritance hierarchy as the added method. The actual change logic is that the developer pulled up the deleted method to a new superclass in the new version. In this way, the old deleted method and the new added method implement the same functionality and thus the mutation changes to the deleted method and the edit for adding the new method may have the same impact to the program. Therefore, using Rule 7, FIFL identifies that both the deleted method and the newly added method cause the same `LC` change: $\texttt{LC}_m$`(CopticChronology, *.getAverageMillisPerMonth())` (i.e., invocation of method `getAverageMillisPerMonth()` on run-time object of type `CopticChronology` resolves to a different method), and thus maps the mutants occurring on the old method with the method addition. In fact, the mutants

for the old method have different test execution results with the new version, making the ranking of the fault-free `AM(getAverageMillisPerMonth())` decreased, and thus the ranking of actual failure-inducing edits increased.

6.3.1.4 Inference for Other Added Elements

For the other added method/field c that neither overrides/hides any existing method/field nor shares the overriding/hiding hierarchy with any deleted method/field, if c is executed by the regression test suite, c must be invoked/accessed by some changed or added method c'. Then, a mutant μ occurring in c' may be used to simulate the impacts of c, because mutant μ in c' may have the same impacts with adding invocation to c in c'. In this situation, the change impact analysis component of FIFL would have detected that change c' depends on change c (i.e., $c \preceq c'$), because c' would invoke/access undeclared method/field without applying change c. Rule 9 then directly maps any mutant that has been mapped with c' to c. Note that Rule 9 should be iteratively applied until it arrives a fix point. To illustrate, the change mapping inference for Figure 6.6(e) is shown as follows:

Applied Rules	$\texttt{AM}(m_2)$	$\texttt{AF}(f_1)$
Rule 9	$Map[\texttt{CM}(m_1)]$	-
Rule 9	$Map[\texttt{CM}(m_1)]$	$Map[\texttt{CM}(m_1)]$

where $Map[\texttt{CM}(m_1)]$ denotes the mutants that have been mapped to change $\texttt{CM}(m_1)$ (Rule 1). The first inference using Rule 9 does not find mapped mutants for $\texttt{AF}(f_1)$, because $\texttt{AM}(m_2)$ has no mapped mutants yet. On the contrary, the second application of Rule 9 successfully finds mapped mutants for $\texttt{AM}(f_1)$,

```
    public XStream(ReflectionProvider...) {
        ...
367:    this.mapper=mapper==null?buildMapper():mapper;
        ...
    }
    private Mapper buildMapper() {
379:    Mapper mapper = new DefaultMapper(classLoaderReference);
380:    if ( useXStream11XmlFriendlyMapper() ){
381:        mapper = new XStream11XmlFMapper(mapper);
382:    }
        ...
    }
```

Figure 6.8: Code snippet of *XStream* V1.21 to illustrate change mapping for `AM` edits without methods sharing the same method overriding hierarchy

because `AM`(m_2) is mapped to mutants at the first step.

Among the studied subjects, when *XStream* evolves from V1.20 to V1.21, the developer added the faulty method `buildMapper()`. Shown in Figure 6.8, the added faulty method `buildMapper()` was invoked by a changed method `XStream()` at line 367 (which is the only changed line for the method, and invoked an old method for building mappers in the old version). The faulty `AM` change is executed by every tests because it is used for initializing mappers, and thus cannot be distinguished from other edits. Because the functionality of building mappers in the old version was also invoked by the changed method, some mutation changes (especially mutation changes on the specific line for invoking the mapper builder) may have similar impacts as adding a new method for building mappers. Thus, FIFL maps the `AM` edit with the mutants that occurred in the old version of `XStream()` based on Rules 1 and 9. In fact, some mutants that occurred at the changed line of the old method `XStream()` exactly fail the same tests with the new version because the state-

ment for constructing the old map builder was changed, demonstrating that mapped mutants can be used to simulate the effect of method additions when the added methods do not override any existing method or share overriding hierarchy with any deleted method.

Finally, Figure 6.6(f) illustrates a slightly more complex situation: an added method m_2 overrides an existing method m_1, and also invokes another added method m_3, which in turns accesses an added field f_1. FIFL first maps $\texttt{AM}(m_2)$ with mutants in m_1 based on Rule 3, then maps $\texttt{AF}(f_1)$ and $\texttt{AM}(m_3)$ with mutants in m_1 based on Rule 9. The detailed inference is shown as follows:

Applied Rules	$\texttt{AM}(m_2)$	$\texttt{AM}(m_3)$	$\texttt{AF}(f_1)$
Rule 3	M_{m_1}	-	-
Rule 9	M_{m_1}	M_{m_1}	-
Rule 9	M_{m_1}	M_{m_1}	M_{m_1}

Note that each program edits will be applied with any applicable rules and may be mapped to mutants from various methods/fields. FIFL uses all those mutants to select the most suitable one (details are shown in the next section).

6.3.2 Mutant Suspiciousness Calculation

The calculation of mutant suspiciousness is based on the intuition that a mutant that has a similar test pass/fail results with the program after edits might share same positions with the real failure-inducing program edits. Therefore, the suspiciousness of a mutant can be calculated based on the similarity between its test execution results and the test execution results after edits.

As mutant suspiciousness will be used to refine edit suspiciousness, we use the same suspicious calculation formulae used by FaultTracer, for calculating the suspiciousness of edits [150]. The difference is that FIFL additionally uses the correlation between tests and mutant killing as input, while FaultTracer only uses the correlation between tests and edits as input. FIFL adapts four representative suspiciousness computations for mutants as follows.

(1) *Statistical Bug Isolation.* Liblit et al. [77] first proposed Statistical Bug Isolation (SBI) to rank faulty predicates. Yu et al. [144] adapted SBI to rank potential faulty statements. FaultTracer adapted the SBI formula to calculate the suspiciousness for program edits [150]. We further adapt the formula to define a suspiciousness score for a mutant μ as $S_s(\mu)$:

$$(\underbrace{|K(\mu,T')\cap T'_f|}_{failed(\mu)})/(\underbrace{|K(\mu,T')\cap T'_p|}_{passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|}_{failed(\mu)})$$

In this formula, T' denotes all affected tests, T'_p denotes the passed affected tests, T'_f denotes the failed affected tests, $K(\mu, T')$ denotes the affected tests that kill μ, $failed(\mu)$ denotes the number of failed affected tests that kill mutant μ, and $passed(\mu)$ denotes the number of passed affected tests that kill μ.

(2) *Tarantula.* Jones et al. [66] proposed Tarantula, which assigns higher suspiciousness scores to statements primarily executed by failed tests than statements primarily executed by passed tests. FaultTracer then adapted the Tarantula formula to calculate the suspiciousness for program edits [150].

Similarly, we adapt the formula to define a suspiciousness score for a mutant μ as $S_t(\mu)$:

$$(\underbrace{|K(\mu,T')\cap T'_f|/|T'_f|}_{\%failed(\mu)})/(\underbrace{|K(\mu,T')\cap T'_p|/|T'_p|}_{\%passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|/|T'_f|}_{\%failed(\mu)})$$

In this formula, $\%failed(\mu)$ denotes the ratio of failed affected tests that can also kill mutant μ to all failed affected tests, while $\%passed(\mu)$ denotes the ratio of passed affected tests that can kill mutant μ to all passed affected tests.

(3) *Ochiai.* Yu et al. [144] and Abreu et al. [8] used the Ochiai formula, which originated from the molecular biology domain, to rank faulty statements in one specific program version. FaultTracer then adapted the Ochiai formula to calculate the suspiciousness for program edits [150]. Similarly, we adapt the formula to define a suspiciousness score for a mutant μ as $S_o(\mu)$:

$$(\underbrace{|K(\mu,T')\cap T'_f|}_{failed(\mu)})/\sqrt{\underbrace{|T'_f|}_{all_failed}*(\underbrace{|K(\mu,T')\cap T'_p|}_{passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|}_{failed(\mu)})}$$

In this formula, all_failed denotes the number of all failed affected tests.

(4) *Jaccard.* Abreu et al. [8] used the Jaccard formula, which was used for measuring the statistical similarity and diversity between sample sets, to rank faulty statements. FaultTracer then adapted the Jaccard formula to calculate the suspiciousness for program edits [150]. We further adapt the formula to define a suspiciousness score for a mutant μ as $S_j(\mu)$:

$$(\underbrace{|K(\mu,T')\cap T'_f|}_{failed(\mu)})/(\underbrace{|T'_f|}_{all_failed}+\underbrace{|K(\mu,T')\cap T'_p|}_{passed(\mu)})$$

In this way, the suspiciousness values for all mutants are between 0.00 and 1.00. If a mutant failed exactly the same set of tests with the program after edits, its suspiciousness would be calculated as 1.00 by all formulae. To illustrate, we show the suspiciousness score calculated for each mutant of the example program in Columns 7-10 of the lower part of Table 6.1. Shown in the gray row, FIFL directly determines a mapping mutant of the real failure-inducing edits as the most suspicious. Note that when a real program has multiple faulty edits, the suspiciousness of mapped mutants for all the faulty edits can be determined as suspicious because the injection of each mutant may make the program fail a subset of the real failed tests. Therefore, all the four formulae can calculate those mutants mapped with faulty edits as suspicious.

6.3.3 Suspiciousness Combination

In this section, we present suspiciousness combination based on the suspiciousness for mutants, the suspiciousness for edits, and the mapping between edits and mutants. FIFL integrates three general combination strategies shown as follows. Note that FIFL uses the maximum suspiciousness value for the set of mapping mutants (i.e., using the most suitable mutant) to refine the suspiciousness of an edit, because a large ratio of mutants might not be effective in simulating the impacts of program edits.

(1) *Min-Max*. This strategy refines an edits's suspiciousness by using the minimum value between the edit's initial suspiciousness value and the maximum suspiciousness value for its mapping mutants:

$$S_{refined}(c) = Min(S(c), Max_{\mu \in Map[c]} S(\mu))$$

To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM(SuperAcnt.deposit())}) = \\ & Min(0.71, Max(1.00, 0.71, 0.71, 0.71)) = 0.71 \end{aligned}$$

The suspiciousness value for other three edits are all refined to 0.00, making `AM(SuperAcnt.deposit())` as the top one suspicious edit.

(2) *Max-Max*. This strategy refines an edits's suspiciousness by FaultTracer using the maximum value between the edit's initial suspiciousness value and the maximum suspiciousness value for its mapping mutants:

$$S_{refined}(c) = Max(S(c), Max_{\mu \in Map[c]} S(\mu))$$

in which $Map[c]$ denotes all the mapping mutants for edit c. To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM(SuperAcnt.deposit())}) = \\ & Max(0.71, Max(1.00, 0.71, 0.71, 0.71)) = 1.00 \end{aligned}$$

The suspiciousness value for other three edits are refined as 0.71, 0.58, and 1.00, making `AM(SuperAcnt.deposit())` tied as the top ranking edit.

(3) *Ratio-Max.* This strategy refines an edit's suspiciousness by assigning different weights to the edit's initial suspiciousness value and the maximum suspiciousness value of its mapping mutants. The combination is shown as follows.

$$S_{refined}(c) = \alpha * S(c) + (1 - \alpha) * Max_{\mu \in Map[c]} S(\mu)$$

where $\alpha \in [0.0, 1.0)$ denotes the weight for an edit's initial suspicious[4]. Note that when $\alpha = 0.0$, the strategy ranks edits only based on the suspiciousness of their mapping mutants. To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation and the default α value of 0.5, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM}(\texttt{SuperAcnt.deposit}())) = \\ & \alpha * 0.71 + \alpha * Max(1.00, 0.71, 0.71, 0.71) \\ & 0.5 * 0.71 + 0.5 * 1.00 = 0.86 \end{aligned}$$

The suspiciousness value for other three edits are refined as 0.36, 0.29, and 0.50, making `AM(SuperAcnt.deposit())` as the top one suspicious edit.

Note that when the number of mapping mutants for an edit is smaller than a threshold (2 in this work), those mutants may not be sufficient to

[4]We exclude the α value of 1.0, because the edit's suspiciousness is not refined at all in such a case.

simulate the impact of the edit. Therefore, for this circumstance, FIFL simply keeps the initial suspiciousness for that edit.

6.3.4 Tackling the Cost of FIFL: Edit-Oriented Mutation Testing

Compared with the existing FaultTracer technique, FIFL additionally utilizes the available mutation testing results of the old program version from the repository to further refine the fault localization results. Mutation testing results can already be available due to its other applications, e.g., generating [34, 42, 49, 95, 156] or evaluating test suites [20, 121, 122]. In addition, since FIFL depends on the mutation testing results of the old version, the mutation testing process can be conducted at the same time with developing the new version, thus improving the fault localization results with no overhead.

However, the mutant testing results for the old program version might still be absent when doing fault localization, we further show how to tackle the cost of mutation testing at that situation. We propose the concept of *Edit-Oriented Mutation Testing*, which only collects mutation testing results of the mutants mapping with program edits, since the execution results of other mutants are not used by FIFL. Formally, the subset of mutants executed by edit-oriented mutation testing can be represented as follows.

$$M_{edit} = \{\mu | \forall c \in \Delta, \mu \in Map[c]\}$$

where Δ denotes all the edits between two program versions, and $Map[c]$ denotes the mapping mutants for program edit c. As the mutant generation is

much more efficient than mutant execution [122], our implementation simply generates all the mutants, and only executes the mutants mapped with edits.

6.4 Implementation

We built our FIFL technique on top of Javalanche [3, 122] and FaultTracer [150]. FIFL uses Javalanche for the first step mutant generation and execution. Javalanche is a state-of-the-art mutation testing tool for Java programs. Javalanche allows efficient mutant generation, as well as mutant execution. More precisely, Javalanche uses a small set of sufficient mutant operators, and manipulates Java bytecode directly using *mutant schemata* to enable efficient mutant generation. In addition, Javalanche only executes the set of influenced tests for each mutant based on coverage checking, and allows parallel execution to enable efficient mutant execution.

FIFL uses FaultTracer for the second step edit detection and suspiciousness calculation. FaultTracer is a state-of-the-art technique for localizing failure-inducing program edits during software evolution. FaultTracer calculates the suspiciousness of each program edit based on their correlation with failed tests. FaultTracer has been shown to outperform the previous ranking technique [109] by more than 50% [150].

FIFL's third step change mapping inference requires the method-overriding hierarchy, field-hiding hierarchy, as well as source code scope for given changed entities, etc. We implemented this step based on the *Eclipse JDT toolkit* [2]. The fourth and the fifth steps mainly involve data computation and trans-

Table 6.2: Subjects overview.

Projects	Description	License	LoC(Src/Test)
TimeMoney	Time and money library	MIT	2.7K/3.0K
Barbecue	Bar-code creator	BSD	5.4K/3.3K
Mime4J	Message stream parser	Apache2.0	7.0K/3.8K
Jaxen	Java XPath library	Apache-style	14.0K/8.8K
Xml-Sec	XML security standards	MIT	19.8K/4.0K
XStream	Object serialization library	BSD	18.4K/20.1K
JMeter	Performance testing	Apache2.0	44.6K*
Com-Lang	Java helper utilities	Apache2.0	23.3K/32.5K
Joda-Time	Time library	Apache2.0	32.9K/55.9K

∗ indicates source and test code are written together, and cannot be measured separately

formation, and are directly implemented with the Java language. Although FIFL is currently implemented for Java programs, the FIFL methodology of localizing faulty edits based on fault injection is generalizable for other object-oriented languages.

6.5 Experimental Study

FIFL aims to make suspiciousness calculation more precise for program edits. To evaluate the effectiveness of FIFL, the experimental study compares FIFL against FaultTracer [150], a state-of-the-art approach for localizing failure-inducing edits on real-world code repositories.

6.5.1 Independent Variables

According to the theory of experimentation, we used the following *independent variables* (IVs) to investigate their influences on the final experimental results:

IV1: Different Localization Approaches. We considered the following

choices of approaches as the first independent variable: (1) FaultTracer, which is a state-of-the-art approach for localizing failure-inducing program edits; (2) FIFL, which is proposed in this chapter and embodies the idea of injecting faults to localize failure-inducing edits.

IV2: Different Calculation Formulae for Edit Suspiciousness. We considered all the four formulae used by FaultTracer [150] to calculate the suspiciousness of program edits: (1) *SBI*; (2) *Tarantula*; (3) *Ochiai*; and (4) *Jaccard*.

IV3: Different Calculation Formulae for Mutant Suspiciousness. Similarly, we considered the same set of formulae for calculating the suspiciousness of mutants (shown in Section 6.3.2): (1) *SBI*; (2) *Tarantula*; (3) *Ochiai*; and (4) *Jaccard*.

IV4: Different Combination Strategies. We considered all the three combination strategies shown in Section 6.3.3 for refining the suspiciousness of edits based on the suspiciousness of mutants: (1)*Min-Max*; (2)*Max-Max*; and (3)*Ratio-Max*. For the *Ratio-Max* strategy, we use values of α ranging from 0.00 to 0.95 with increments of 0.05, i.e., 20 values of α. Note that when α=0.0, the strategy ranks edits based on pure mutant suspiciousness.

6.5.2 Dependent Variables

Since we are concerned with the effectiveness as well as efficiency achieved by our FIFL approach, we used the following dependent variable (DV):

DV: Rank of Failure-Inducing Edits. This variable denotes the total

number of edits that developers need to inspect before finding the real failure-inducing edits when using the compared techniques.

6.5.3 Subjects and Experimental Setup

We obtained versions of the source code of nine open-source projects in various application domains, which have been widely used for regression testing and mutation testing research [35,121,122,155]. Table 6.2 depicts brief information about the latest release of each studied project. The sizes of the studied projects range from 5,675 lines of code (LoC) (*TimeMoney*, including 2,678 LoC source code and 2,997 LoC test code) to 88,835 LoC (*Joda-Time*, with 32,932 LoC source code and 55,903 LoC test code). We obtained *Xml-Sec* and *JMeter* from the well-known Software-artifact Infrastructure Repository (SIR) [35], and all the other projects from their host repositories. For each project, we obtained all the available releases in its repository, and treated every two continuous releases as a version pair. For each version pair, we applied the regression test suite of the old version on the new version, and treated the edits that cause the regression suite to fail on the new version as regression faults. We studied all those version pairs with regression test failures to evaluate FIFL's performance. The experimental study was performed on a Dell desktop with Intel i7 8-Core Processor (2.8G Hz), 8G RAM, and Win7 Enterprise 64-bit version.

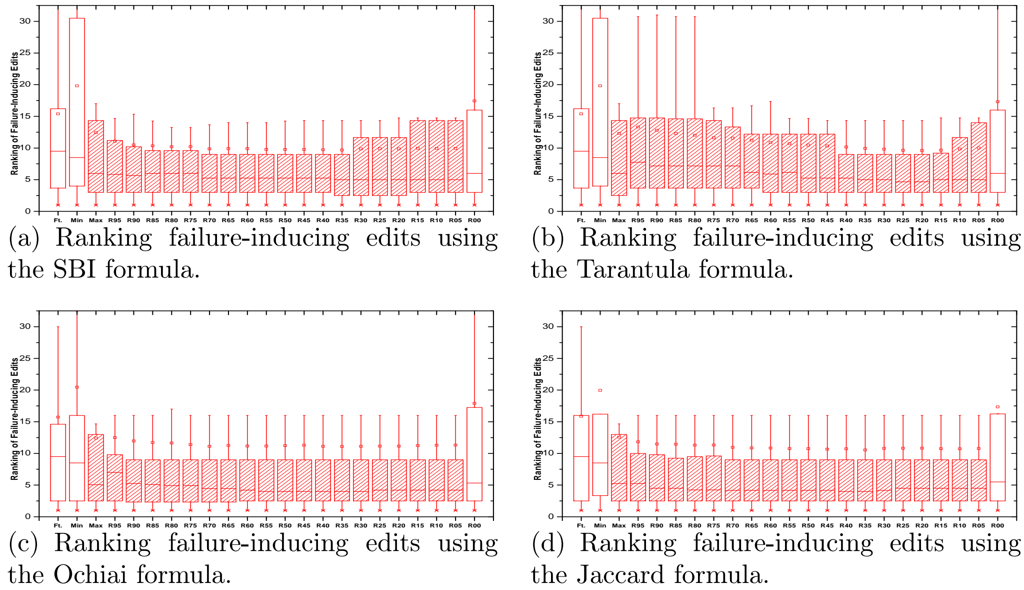
For all the projects, we were able to find version pairs with regression faults except *JMeter*. However, *JMeter* comes with seeded faults in SIR,

Table 6.3: Version pairs with test failures.

No	Project	Version Pair	#Tests	#FTests	#Edits	#FEdits	All Mutants		Mapped Mutants	
							Number	Execution Time	Number	Execution Time
P_1	*TimeMoney*	3.0, 4.0	143	1	215	1	1737	9min24s	792	7min38s
P_2	*TimeMoney*	4.0, 5.0	159	1	246	1	1984	6min43s	325	1min5s
P_3	*Barbecue*	1.5a1, 1.5a2	160	2	23	1	41310	15min37s	96	57s
P_4	*Mime4J*	0.50, 0.60	120	8	2862	3	19111	181min20s	8086	37min50s
P_5	*Mime4J*	0.61, 0.70	348	3	3160	4	27654	227min14s	24443	49min15s
P_6	*Jaxen*	1.0b7, 1.0b9	24	2	204	3	3820	100min25s	964	28min38s
P_7	*Jaxen*	1.1b2, 1.1b5	69	2	419	1	5489	19min8s	1493	7min42s
P_8	*Jaxen*	1.1b6, 1.1b7	243	2	473	5	9704	44min49s	6037	24min8s
P_9	*Jaxen*	1.1b9, 1.1b11	645	1	92	1	10045	61min59s	157	2min9s
P_{10}	*Xml-Sec*	1.0, 2.0	91	5	329	2	10599	16min6s	1134	2min3s
P_{11}	*XStream*	1.20, 1.21	637	3	209	1	10956	49min51s	547	5min31s
P_{12}	*XStream*	1.21, 1.22	698	1	222	2	11516	54min24s	847	6min56s
P_{13}	*XStream*	1.22, 1.30	768	24	540	11	12536	64min22s	1870	8min25s
P_{14}	*XStream*	1.30, 1.31	885	12	416	3	14140	96min43s	2206	13min0s
P_{15}	*XStream*	1.31, 1.40	924	13	1225	7	15006	99min26s	3462	22min15s
P_{16}	*XStream*	1.41, 1.42	1200	6	136	5	18046	132min2s	1817	10min50s
P_{17}	*JMeter*	0.0, 1.0	51	1	1714	1	5363	29min56s	1779	8min40s
P_{18}	*JMeter*	1.0, 2.0	60	2	1056	1	21896	57min32s	3604	13min22s
P_{19}	*JMeter*	2.0, 3.0	72	11	2809	4	8067	43min27s	4347	34min44s
P_{20}	*JMeter*	3.0, 4.0	76	1	764	1	7116	34min36s	703	10min50s
P_{21}	*Com-Lang*	3.02, 3.03	1698	1	221	1	20792	90min26s	803	4min7s
P_{22}	*Com-Lang*	3.03, 3.04	1703	2	172	2	20792	90min29s	1441	3min33s
P_{23}	*Joda-Time*	0.90, 0.95	219	4	5976	2	6581	6min29s	5365	3min37s
P_{24}	*Joda-Time*	0.98, 0.99	1932	6	1254	2	16208	31min36s	3631	4min45s
P_{25}	*Joda-Time*	1.10, 1.20	2420	1	793	1	19012	53min10s	1997	12min32s
P_{26}	*Joda-Time*	1.20, 1.30	2516	11	571	3	19566	106min10s	718	31min32s

thus we use seeded faults for *JMeter*. In total, we have 26 version pairs with regression faults, and the details are shown in Table 6.3. In Table 6.3, Column 1 shows the abbreviations for all the version pairs with regression faults. Columns 2 and 3 show the project name and corresponding versions for each version pair. Columns 4 and 5 show the number of tests and failed tests for each version pair. Columns 6 and 7 show the number of edits and failure-inducing edits for each version pair.

The table also presents the mutation testing statistics using Javalanche. Columns 8 and 9 show the number and execution time for all mutants. Similarly, Columns 10 and 11 show the number and execution time for the mutants that are actually needed by FIFL (i.e., mutants mapped with edits). We observe that overall mutation testing time by Javalanche for each studied version pair is acceptable for the studied subjects, ranging from 6 minutes 43 seconds to 227 minutes 14 seconds. The reason is that Javalanche embodies a set of optimization strategies for improving efficiency (Section 6.4). We also find that the mutation testing time for only mapped mutants is much more efficient, and is less than 1 hour for all subjects. Recall that FIFL never costs the developer the entire mutation testing time: (1) mutation testing results can already be in the repository before applying FIFL due to its other applications; (2) the mutation testing results can be collected at the same time as developing the new version, because FIFL uses the mutation testing results on the old version; (3) even when mutation testing results are not available before applying FIFL, developers can only collect the mutation testing results



(a) Ranking failure-inducing edits using the SBI formula.

(b) Ranking failure-inducing edits using the Tarantula formula.

(c) Ranking failure-inducing edits using the Ochiai formula.

(d) Ranking failure-inducing edits using the Jaccard formula.

Figure 6.9: Ranking failure-inducing edits using various techniques with various formulae.

for the mutants mapped with edits (Section 6.3.4).

6.5.4 Results and Analysis

In this section, we first compare all strategies of FIFL with FaultTracer (Section 6.5.4.1). Then we compare the default strategies of FIFL with FaultTracer in detail (Section 6.5.4.2). Finally, we discuss about the scope and limitations of the FIFL approach and its evaluation (Section 6.5.4.3).

6.5.4.1 Overall comparison between FaultTracer and various strategies of FIFL

Figures 6.9(a) to 6.9(d) show the comparison of FIFL with FaultTracer for different suspicious calculation formulae. We denote FaultTracer (the rank-

ing based on pure edit suspiciousness) as *Ft.*, FIFL's *Min-Max* strategy as *Min*, and FIFL's *Max-Max* strategy as *Max*. For FIFL's *Ratio-Max* strategies, we use R and the value of α to represent each strategy. For example, we use *R95* to denote the *Ratio-Max* strategy with α=0.95. In each figure, the horizontal axis shows compared techniques, and the vertical axis shows the rank of failure-inducing edits by each technique across all the version pairs with regression faults. Each box plot shows the average (a dot in the box), median (a line in the box), and upper/lower quartile values for the ranking of failure-inducing edits across various version pairs[5]. We mark all the techniques that outperform the original FaultTracer technique (which is based on the pure edit suspiciousness) in terms of both average and median values as shadowed box plots. The key findings from the experimental results are as follows.

First, in terms of median effectiveness across all version pairs, all ranking techniques of FIFL outperform FaultTracer. When using the SBI formula (the median case for the Tarantula formula is similar), in the median case, FaultTracer localizes failure-inducing edits within 9.5 edits. In contrast, the *Min-Max* and *Max-Max* strategies of FIFL localize failure-inducing edits within 8.5 and 6 edits, respectively. The *Ratio-Max* strategies of FIFL with all α values within $[0.00, 0.95]$ localize failure-inducing edits within 6 edits. When using the Ochiai formula (the median case for the Jaccard formula is similar), in the median case, FaultTracer localizes failure-inducing edits within

[5]Note that for each version pair with multiple faulty edits, we use the average ranking of all its faulty edits.

9.5 edits. In contrast, the *Min-Max* and *Max-Max* strategies of FIFL localize failure-inducing edits within 8.5 and 5.05 edits, respectively. Furthermore, the *Ratio-Max* strategies of FIFL with $\alpha \in [0.30, 0.55]$ localize failure-inducing edits within 4 edits, an improvement of 57.89% over FaultTracer.

Second, in terms of average effectiveness across all version pairs, the *Max-Max* strategy and the *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ still outperform FaultTracer. For example, using the SBI formula, FaultTracer localizes failure-inducing edits within 15.40 edits, the *Max-Max* strategy of FIFL localizes failure-inducing edits within 12.42 edits, and all strategies of FIFL with $\alpha \in [0.05, 0.95]$ are able to localize faulty edits within 11.08 edits. Furthermore, the *Ratio-Max* strategy of FIFL with $\alpha = 0.35$ localizes failure-inducing edits within 9.68 edits, indicating an average improvement of 37.14% over FaultTracer. However, the *Min-Max* strategy and the *Ratio-Max* strategy with $\alpha = 0.00$ (the ranking based on pure mutant suspiciousness) cannot outperform FaultTracer. For example, when using the SBI formula, the *Min-Max* strategy and the *Ratio-Max* strategy with $\alpha = 0.00$ (the ranking based on pure mutant suspiciousness) localize failure-inducing edits within 19.81 and 17.43 edits, respectively. The reason is that for some failure-inducing edits, accidentally none mapped mutant can simulate their real impacts. Then the suspiciousness values for their mapped mutants can be quite low (even 0.00), making the *Min-Max* strategy and the strategy based on pure mutant suspiciousness perform extremely worse at those cases.

Third, in terms of stability, the *Max-Max* strategy and the *Ratio-Max*

strategies with $\alpha \in [0.05, 0.95]$ outperform FaultTracer. As Figures 6.9(a) to 6.9(d) show, the box plots representing *Max-Max* and *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ are consistently more condensed than that of FaultTracer based on pure edit suspiciousness. To validate this observation, we also compute the Standard Deviations (SD) for each compared technique. The SDs for the *Max-Max* strategy and all *Ratio-Max* strategies with α from 0.05 to 0.95 are also consistently smaller than that of FaultTracer across all the four formulae.

Fourth, for different formulae, different α values have different impacts for the *Ratio-Max* strategy. For example, all α values perform similarly for both the Ochiai and Jaccard formulae. On the contrary, α values between 0.35 and 0.85 perform better than other values for the SBI formula, and α values between 0.15 and 0.40 perform better than other values for the Tarantula formula. An interesting finding is that even adding a little flavor of mutant suspiciousness to the edit suspiciousness (i.e., α=0.95) would boost the ranking based on pure edit suspiciousness (i.e., FaultTracer) significantly. For example, when using the Jaccard formula, in the median case, FaultTracer is able to localize faults within 9.5 edits, while the *Ratio-Max* strategy with α=0.95 is able to localize faults within 5.25 edits, thus significantly reducing the burden on developers to localize faults.

Finally, we also perform statistical tests to compare FaultTracer with

various FIFL strategies[6]. For each FIFL strategy, we use its ranking of failure-inducing edits on different version pairs as a sample data set, and compare it against the corresponding sample set for FaultTracer. Before applying paired significance test, we first apply the *Shapiro-Wilk Normality Test* [124] to check the normality assumption. The results show that the differences between any FIFL strategy and FaultTracer do not follow normal distribution even at the 0.01 significance level. Therefore, we choose to use the *Wilcoxon Signed-Rank Test* [133] to compare FIFL and FaultTracer, because it is suitable for the case that the sample differences may not be normally distributed [81]. Table 6.4 shows the detailed Wilcoxon test results.

In Table 6.4, Column 1 shows the various FIFL strategies compared against FaultTracer. Columns 2-5 show the p values for comparing the corresponding FIFL strategies with FaultTracer when using the four different formulae. The *Null* hypothesis was rejected at the 0.01 significance level (i.e., $p < 0.01$) when comparing all FIFL *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ against FaultTracer, indicating that the vast majority of FIFL strategies are able to statistically (i.e., not likely to be accidentally) outperform FaultTracer in localizing failure-inducing edits. The table also shows that the *Null* hypothesis was not rejected at the 0.01 significance level when comparing FaultTracer against FIFL's *Min-Max* strategy, *Max-Max* strategy, and the strategy based on pure mutant suspiciousness (i.e., *Ratio-Max* strategy with α value of 0.00), indicating that these three FIFL strategies may not outperform FaultTracer

[6] All the statistical tests used in this work were performed using the *R* language [54].

consistently. One interesting finding is that although the FIFL *Max-Max* strategy is able to outperform some FIFL *Ratio-Max* strategy with $\alpha \in [0.05, 0.95]$ in terms of average/median performance, the *Max-Max* strategy is not able to outperform FaultTracer in terms of significance tests while all FIFL *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ outperform FaultTracer. The reason is that the performance of the *Max-Max* strategy is not stable for different subjects – it outperforms FaultTracer substantially for some subjects but also performs worse than FaultTracer for some subjects. On the contrary, although some *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ cannot outperform FaultTracer substantially, it outperforms FaultTracer consistently across different subjects. The results demonstrate that using both edit suspiciousness and mutant suspiciousness for ranking each edit (i.e., FIFL's *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$) performs better than using either edit suspiciousness or mutant suspiciousness for ranking each edit (i.e., FaultTracer that uses pure edit suspiciousness, FIFL *Min-Max* that uses the lower suspiciousness values, FIFL *Max-Max* that uses the higher suspiciousness values, and FIFL *Ratio-Max* with $\alpha = 0.00$ that uses pure mutant suspiciousness). The reason is that both the spectrum information and the impact information (simulated by mutation testing) are useful for localizing failure-inducing edits, and thus using any one of them for one edit may not be both precise and stable.

In summary, both the descriptive statistics and the significance tests show that a vast majority of FIFL strategies are able to outperform FaultTracer significantly. Furthermore, the significance tests show that using both

Table 6.4: Wilcoxon tests for comparing FIFL techniques with FaultTracer

FIFL	SBI (p)	Tarantula (p)	Ochiai (p)	Jaccard (p)
Min.	0.1179	0.1179	0.6602	0.2679
Max.	0.1487	0.1147	0.1089	0.1207
R95	0.0006**	0.0011**	0.0011**	0.0006**
R90	0.0006**	0.0007**	0.0007**	0.0007**
R85	0.0006**	0.0011**	0.0004**	0.0007**
R80	0.0006**	0.0009**	0.0012**	0.0006**
R75	0.0006**	0.0008**	0.0006**	0.0010**
R70	0.0007**	0.0008**	0.0006**	0.0011**
R65	0.0006**	0.0008**	0.0006**	0.0013**
R60	0.0006**	0.0012**	0.0007**	0.0013**
R55	0.0004**	0.0006**	0.0004**	0.0008**
R50	0.0004**	0.0004**	0.0005**	0.0009**
R45	0.0004**	0.0004**	0.0006**	0.0008**
R40	0.0004**	0.0004**	0.0005**	0.0017**
R35	0.0004**	0.0004**	0.0008**	0.0010**
R30	0.0013**	0.0004**	0.0009**	0.0035**
R25	0.0013**	0.0004**	0.0021**	0.0035**
R20	0.0013**	0.0004**	0.0024**	0.0035**
R15	0.0014**	0.0004**	0.0038**	0.0040**
R10	0.0014**	0.0013**	0.0040**	0.0040**
R05	0.0014**	0.0012**	0.0045**	0.0040**
R00	0.4665	0.4444	0.4079	0.2762

∗ indicates significance at the 0.05 level ($p<0.05$)
∗∗ indicates significance at the 0.01 level ($p<0.01$)

edit suspiciousness and mutant suspiciousness for ranking each edit performs better than using either edit suspiciousness or mutant suspiciousness for ranking each edit, further demonstrating the motivation of the work – combining edit spectrum information and edit impact information (simulated by mutation testing) can achieve better fault localization results.

6.5.4.2 Detailed comparison between FaultTracer and FIFL with the default settings

We present the detailed comparison between FaultTracer and FIFL's *Ratio-Max* strategy with the default α of 0.50 on all the version pairs. In Table 6.5, Column 1 lists all the version pairs with regression faults. Columns

2-4 present the average rank of faulty edits by FaultTracer, the average rank of faulty edits by FIFL, and improvement by FIFL over FaultTracer(%) using the SBI formula for each subject. Note that we also show the improvement achieved by FIFL without mapping approximations for addition edits (Rules 3-9 in Figure 6.3) in parentheses. Similarly, Columns 5-13 present the comparison between FaultTracer and FIFL using the Tarantula, Ochiai, and Jaccard formulae.

In general, using all the formulae, all FIFL techniques in Table 6.5 are able to achieve improvements over FaultTracer for the majority of the version pairs. Also, the statistical test in Table 6.4 also confirms that all FIFL techniques in Table 6.5 can statistically outperform FaultTracer. For example, using the default SBI formula, FIFL outperforms FaultTracer by 2.33% to 86.26% for 16 of 26 version pairs and is only slight inferior than FaultTracer on one version pair (with an average improvement of 36.46%). The reason FIFL techniques in Table 6.5 outperform FaultTracer over the vast majority of the studied subjects is that those FIFL techniques use both the edit suspiciousness and mutant suspiciousness for ranking each edit, which provide both coverage and impact information for precise fault localization. The reason FIFL techniques do not outperform FaultTracer on every case is that FaultTracer techniques already rank the failure-inducing edits precisely using only edit suspiciousness for some version pairs, and the use of mutant suspiciousness may bring some noises to the ranked list. The experimental data supports this reasoning: for example, among the 10 version pairs where

FIFL cannot outperform FaultTracer using the SBI formula, FaultTracer is already able to localize failure-inducing edits within 5 edits for 8 version pairs, leaving little room for FIFL to improve.

We also observe that even FIFL without mapping approximations is also able to outperform FaultTracer significantly. For example, using the SBI formula, FIFL without mapping approximations can outperform FaultTracer for 14 of 26 version pairs with an average improvement of 30.72%. For some version pairs (e.g., P_{14}), FIFL without mapping approximations even slightly outperforms FIFL with mapping approximations. The reason is that FIFL without mapping approximations aggressively ignores the chance to increase the suspiciousness for addition edits using mutant suspiciousness, and thus performing better for some version pairs with only faults in non-addition edits. However, for the majority of the version pairs, FIFL with mapping approximations performs better.

To further understand the performance of FIFL, we also manually analyzed why FIFL outperform or cannot outperform FaultTracer for each subject using the SBI formula. We describe the following interesting cases:

Case 1. When *XStream* evolved from V1.20 to V1.21 (P_{11}), 3 tests failed because the developers added faulty method `XStream.buildMapper()` (shown in Figure 6.8), which is used to initialize `XStream` object and is executed by every test. Therefore, FaultTracer, which treats edits mainly executed by failed tests as more suspicious, cannot rank this `AM` edit high. FIFL without mapping approximations cannot improve over FaultTracer because it cannot

Table 6.5: Comparison between FaultTracer and default settings of FIFL.

Rev.	SBI				Tarantula				Ochiai				Jaccard			
	Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)	
P_1	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)
P_2	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_3	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_4	6.33	6.33	0.00	(0.00)	6.33	6.33	0.00	(0.00)	2.33	2.33	0.00	(0.00)	2.33	2.33	0.00	(0.00)
P_5	25.00	14.25	43.00	(23.00)	25.00	13.50	46.00	(23.00)	12.25	12.00	2.04	(4.08)	11.00	12.25	-11.36	(-9.09)
P_6	16.00	14.00	12.50	(25.00)	16.00	14.67	8.33	(20.83)	16.00	14.33	10.42	(22.92)	16.00	14.33	10.42	(22.92)
P_7	34.00	32.00	5.88	(2.94)	34.00	32.00	5.88	(2.94)	30.00	16.00	46.67	(23.33)	30.00	16.00	46.67	(23.33)
P_8	16.20	8.40	48.15	(48.15)	16.20	12.20	24.69	(24.69)	14.60	8.40	42.47	(42.47)	16.20	8.40	48.15	(48.15)
P_9	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_{10}	5.50	4.50	18.18	(0.00)	5.50	4.00	27.27	(0.00)	5.00	4.00	20.00	(0.00)	5.00	4.50	10.00	(0.00)
P_{11}	13.00	3.00	76.92	(0.00)	13.00	4.00	69.23	(15.38)	13.00	3.00	76.92	(0.00)	13.00	3.00	76.92	(0.00)
P_{12}	7.00	4.00	42.86	(42.86)	7.00	4.50	35.71	(35.71)	7.00	4.00	42.86	(42.86)	7.00	4.00	42.86	(42.86)
P_{13}	59.45	37.36	37.16	(22.94)	59.45	42.45	28.59	(14.37)	78.73	63.36	19.52	(18.24)	73.64	59.45	19.26	(14.44)
P_{14}	43.67	6.00	86.26	(93.13)	43.67	10.00	77.10	(83.21)	42.00	5.67	86.51	(92.06)	44.00	4.33	90.15	(94.70)
P_{15}	62.86	39.43	37.27	(43.64)	62.86	43.43	30.91	(33.64)	66.00	62.14	5.84	(7.58)	64.86	48.00	25.99	(29.52)
P_{16}	15.40	6.60	57.14	(36.36)	15.40	6.40	58.44	(37.66)	12.20	4.40	63.93	(34.43)	15.40	6.60	57.14	(36.36)
P_{17}	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)
P_{18}	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)
P_{19}	32.25	31.50	2.33	(2.33)	32.25	31.00	3.88	(3.10)	48.50	47.00	3.09	(2.58)	51.00	49.75	2.45	(4.41)
P_{20}	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)
P_{21}	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)
P_{22}	2.50	2.50	0.00	(0.00)	2.50	3.00	-20.00	(-20.00)	2.50	2.50	0.00	(0.00)	2.50	2.50	0.00	(0.00)
P_{23}	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)
P_{24}	3.00	3.00	0.00	(0.00)	3.00	3.00	0.00	(0.00)	2.50	2.50	0.00	(0.00)	2.50	2.50	0.00	(0.00)
P_{25}	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)
P_{26}	3.67	4.00	-9.09	(-18.18)	3.67	3.67	0.00	(-9.09)	2.00	2.67	-33.33	(-33.33)	3.67	3.67	0.00	(-9.09)
Avg	15.40	9.78	36.46	(30.72)	15.40	10.45	32.14	(26.12)	15.74	11.22	28.67	(23.75)	15.87	10.74	32.35	(27.49)

map the addition edit to any mutant. In contrast, FIFL with mapping approximations maps the edit to mutants inside changed method `XStream.XStream()` using Rules 1 and 9. Some mapped mutants failed exactly the same set of tests with the new program version because those mutants mutate the old statements for building mappers inside `XStream.XStream()`, and therefore boost the ranking of the failure-inducing edit by 76.92%.

Case 2. When *Com-Lang* evolved from V3.02 to V3.03 (P_{14}), test `FastDateFormatTest.testLang538` failed because the developer removed a conditional block for updating time zone in method `FastDateFormat.format()` (shown in Figure 6.4). The changed method is used by 35 tests that involve date format transformation. However, only 1 of them failed because the other 34 tests do not check the detailed time zone, making `CM(FastDateFormat.format())` have a suspiciousness value of only 0.0286 using FaultTracer, and not able to be ranked high. In contrast, using Rule 1, FIFL maps `CM(FastDateFormat.format())` with 5 mutants (each mapped with a line in the method in V3.02), three of which are killed exactly by the failed test and thus have suspiciousness values of 1.0. In this way, FIFL precisely localizes the failure-inducing program edit within top 2 edits, outperforming FaultTracer by 80%. In this case, FIFL without mapping approximation can also localize the fault precisely because the failure-inducing edits is not addition change.

Case 3. When *JMeter* evolved from V1.0 to V2.0, test `testArgumentCreation` in class `ArgumentsPanel.Test` and test `testTreeConversion` in class `Save.Test` failed because of one faulty edit, `AM(NamePanel.updateName())`. Although

the suspiciousness value of the edit is already 1.0 using FaultTracer, 12 other edits also have the suspiciousness value of 1.0, making FaultTracer only rank the failure-inducing edit within top 13 edits. In contrast, FIFL is able to localize the failure-inducing edit within 9 edits because FIFL refines the suspiciousness of each edit based on their mapping mutants, and decreases the suspiciousness values of 4 top-ranked fault-free edits.

Case 4. When *Mime4J* evolved from V0.61 to V0.70, 3 tests failed because of 4 failure-inducing program edits. FaultTracer and FIFL perform similarly on 2 failure-inducing edits. The other 2 failure-inducing edits are `CM` and `AM` edits on constructors of `MimeBoundaryInputStream` class. Because the 2 failure-inducing edits are constructor edits, they are executed by almost all tests, making FaultTracer only localize them within 25 edits. In contrast, FIFL maps the two constructor edits with mutants using Rule 1 and Rule 3, respectively. Some mapped mutants have the suspiciousness value of 1.0 because they are only killed by one failed test, making FIFL able to localize the two failure-inducing edits within top 2 edits.

Case 5. When *Xml-Sec* evolved from V1.0 to V2.0 (P_{10}), there are two failure-inducing edits: `AM` on method `engineCanonicalizeXPathNodeSet()` inside the class `CanonicalizerBase`, and `CM` on `circumventBug2650()` in class `XMLUtils`. Using Rule 7, FIFL finds that mutants in the deleted method `engineCanonicalizeXPathNodeSet()` of class `Canonicalizer20010315` are able to simulate the impacts of the newly added method. The mapped mutants increase the rank of the failure-inducing `AM` edit by 1. In addition,

AF(CanonicalizerBase._includeComments) can also be mapped with mutants inside a deleted method by applying Rules 7 and 9. Then, AF edit's rank was lowered correspondingly, making the rank of the failure-inducing CM edit, CM(XMLUtils.circumventBug2650()), increased by 1. Therefore, the average ranking for two failure-inducing edits is improved by 18.18%.

Case 6. The evolution from *Joda-Time*1.20 to 1.30 (P_{26}) is the only case where FaultTracer outperforms FIFL using SBI. The reason is that one failure-inducing edit, CSFI(GregorianChronology.MAX_YEAR), does not have mapped mutants, and another fault-free edit, CM(BasicChro- nology.getYear()), has mapped mutants that accidentally share some failed tests with the new version.

In summary, the *Ratio-Max* strategy of FIFL with default setting is able to outperform FaultTracer significantly. For example, even the default setting of FIFL with SBI formula outperforms FaultTracer by 2.33% to 86.26% on 16 of 26 studied version pairs, and is only inferior than FaultTracer on one version pair.

6.5.4.3 Discussions

Although automated fault localization approaches [8,48,66,77,107,144–146, 150, 163] have been intensively studied for more than a decade, there are common limitations for them. For example, Parnin and Orso [105] recently argued that existing fault localization approaches rely on a strong assumption that examining a potential faulty statement in isolation is enough to localize

and fix a fault. They performed a case study showing that a traditional fault localization technique (which ranks all program statements to localize faults) does not help the developer much with localizing faults. The study shows that traditional fault localization at the statement granularity can be painful because (1) it may cause the developer to inspect a extremely long ranked list for large program and (2) developers tend to also inspect the context of each statement other than the statement itself. Therefore, the study suggested that fault localization at the method or file granularities may be a promising direction for fault localization, because those granularities provide a shorter candidate list and provide enough context information for each ranked entity. Although our FIFL approach may share the same limitations with traditional fault localization, FIFL makes attempts to address the limitations of traditional fault localization. For example, FIFL focuses on program edits during software evolution, which provides a much shorter ranked list than ranking all program statements. In addition, FIFL extracts program edits at the method/field level, which provide the developer enough context information for reasoning each ranked entity.

There is also another intrinsic limitation for the spectrum-based fault localization approaches that FIFL is based on [8,66,77,144,150] – they only use the correlation between the coverage of program elements with test pass/fail results to localize faults. However, there is a gap between the coverage and the actual impact of program elements to the test pass/fail results. In this work, we make an attempt to bridge the gap by using the simulated impact

Table 6.6: Summary results when using the default *R50* strategy to rank all edits and rank edits for each failed test

R50 Formula	Rank all edits			Rank edits per test		
	Ft.	Fi.	Impr.	Ft.	Fi.	Impr.
SBI	15.40	9.78	36.46%	10.12	5.83	42.44%
Tarantula	15.40	10.45	32.14%	10.12	6.10	39.70%
Ochiai	15.74	11.22	28.67%	10.43	6.21	40.48%
Jaccard	15.87	10.74	32.35%	10.80	6.00	44.47%

information via mutation testing. However, the impact information simulated by mutation testing may be still not precise enough. In addition, some edits may not even have mapped mutants due to various reasons, e.g., the mutation operators do not support the specific statement pattern. In the future, we hope more research efforts can be put into this area to bridge the gap between program coverage information and actual impact information using more advanced techniques.

Our experimental evaluation also has limitations. In this work, we used FaultTracer and FIFL to directly rank all the edits once for all failed tests. This corresponds to the debugging process that the developer iterates over all the edits to find all the potential faults for the failed program version. However, some deveopers may prefer to inspect related edits for each failed test to fix failed tests one by one. Therefore, we also used FaultTracer and FIFL to rank edits related to each failed test (i.e., only ranking the edits that are affecting changes of each failed test based on their suspiciousness). In this case, we would have a ranked list of edits for each failed test. For each subject, we collected the average rank of each failure-inducing edit on each failed test. Table 6.6 shows the average summary results across all subjects for FaultTracer

and FIFL's default strategy when ranking all edits together and when ranking edits for each failed test. In the table, Column 1 presents the four formulae. Columns 2-4 show the average rank of failure-inducing edits by FaultTracer and FIFL when ranking all edits together, and the improvement of FIFL over FaultTracer. Similarly, Columns 5-7 present the comparison between FIFL and FaultTracer when ranking edits for each failed test. We can observe that FIFL outperforms FaultTracer even more when ranking edits for each failed test. For example, when using the default strategy with the SBI formula, FIFL is able to localize failure-inducing edits within 5.83 edits for each failed test, indicating an improvement of more than 40% over FaultTracer.

Last but not least, Murphy-Hill et al. [91] recently presented an interesting study showing a suite of factors that can cause a program fault to be fixed in different ways at different circumstances or time points. One such factor is the development phase of the project. For example , when fixing a fault at an earlier phase, developers may choose to fix the root cause of the fault so that if a risk raises, they would have a longer period to compensate. On the contrary, when fixing a fault at a later phase, developers may choose to make a walk-around which would be "least disruptive". The findings in this study raises serious questions for traditional bug prediction [69, 165] or fault localization techniques [8, 66, 77, 144, 150], because faults can actually be fixed in different ways and locations rather than the root causes. The effectiveness evaluation of FIFL may also be influenced, because we only evaluate FIFL in localizing the root cause of failure-inducing edits, whereas the developer

may choose to fix the faults in different ways at different program locations. However, we believe that FIFL may still be useful for developers even when they finally decide not to fix the faults at the root-cause locations, because fully understanding of the fault root cause is still preferred no matter where the faults are finally fixed (also confirmed by the same study [91]).

6.5.5 Threats to Validity

Threats to internal validity are concerned with uncontrolled factors that may also be responsible for the results. In this work, the main threat to internal validity is the possible faults in the implementation of the compared techniques. To reduce this threat, we built FIFL on top of state-of-the-art tools [122, 150], and implemented FIFL using well-known libraries such as *Eclipse JDT toolkit* and *ASM bytecode manipulation framework*. We also reviewed all the code that we produced to assure its correctness. The first author, with Java programming experience for eight years, isolated the failure-inducing edits manually. We have also reviewed all outputs produced by FIFL manually to ensure correctness. However, because this inspection was done manually, there is still a risk of introducing subjectivity and errors.

Threats to external validity are concerned with whether the findings in our study are generalizable for other situations. To mitigate threats to external validity, we used all released versions of nine medium sized open source projects from various application areas. In addition, as our work is related to both regression testing and mutation testing, we also ensure that

the selected subjects have been used for regression testing or mutation testing research [27,28,35,88,121,122,148,150,155,160,161]. However, they still might be not representative for all the possible subject programs.

Threats to construct validity are concerned with whether the measurement in our study reflects real-world situations. To mitigate threats to construct validity, we measured the ranking of failure-inducing edits, which denotes the number of edits that the developer need to manually inspect before finding the fault. Furthermore, we also compare FIFL with the existing technique for localizing failure-inducing edits (FaultTracer [150]) in the same experimental setting. The ranking of failure-inducing program elements has been widely used in the fault localization research area [8, 66, 105, 109, 110, 144, 150]. However, the ranking of failure-inducing program elements may still not correlate with the actual costs in inspecting those elements. To further reduce this threat, inspired by user case studies in other areas [72, 86], rigorous and well-designed studies for investigating the correlation between the ranking of faulty elements and actual fault localization costs should be performed in future work. In addition, as recent work has shown that it is suitable to use program repair techniques to evaluate fault localization techniques fully automatically [107], we also plan to use automated program repair techniques [75, 132] to further demonstrate the effectiveness of FIFL.

6.6 Summary

This chapter makes the following contributions:

- We **unify two widely used dimensions of software changes**: mechanical mutation changes and developer edits. This chapter leverages this unified view to calculate the spectra as well as impacts of program edits to localize faults for evolving software. Furthermore, this unified view can also impact other realms of software testing.

- We **present the FIFL fault localization framework** to improve the accuracy of state-of-the-art techniques for localizing failure-inducing edits using the existing mutation testing results on the old program version. This framework creates a new dimension of possibilities to improve fault localization during software evolution.

- We **present an empirical study** on the code repositories of nine real-world Java programs. The experimental results show that FIFL (using its default settings) is able to outperform the state-of-the-art FaultTracer technique significantly (e.g., by more than 80% for some subjects) in localizing failure-inducing edits, indicating a promising future for localizing faulty edits by injecting mechanical faults.

Chapter 7

Related Work

7.1 Regression Testing

Regression testing [39, 47, 50, 51, 67, 102, 114, 117, 142, 148, 159], which aims to efficiently and effectively run the regression test suites on new program versions, mainly consists of three areas: test selection, test prioritization, and test reduction. Table 7.1 shows these areas and their applications to the mutation testing area.

7.1.1 Test Selection

Test selection determines a subset of tests which have been influenced by program changes and need to be rerun on the new program version. Rothermel et al. [114] identified the differences between two program versions as dangerous edges on control-flow graphs, and only incrementally re-executed the subset of tests whose behaviors might have been influenced by the danger-

Table 7.1: Regression testing areas and their applications for mutation testing

	Regression Testing	Mutation Testing
Test Selection	Others [51, 102, 114], Us [150–152]	Us [155]
Test Prioritization	Others [39, 116], Us [88, 148, 159]	Us [153, 155]
Test Reduction	Others [18, 22, 47, 50], Us [154]	Us [153]

ous edges. Harrold et al. [51] then extended the test selection technique with object-oriented features and applied it to Java programs. Orso et al. [102] further proposed to apply test selection in two phases: (1) compute a coarser-level graph analysis, (2) perform finer-level graph analysis only when needed. This two-phase approach scaled test selection for larger Java programs. Traditional change impact analysis techniques [109, 110, 118] also contain the test selection methodology. Compared with traditional test selection, change impact analysis techniques apply test selection at a coaser granularity, i.e., at the level of method changes and field changes. As shown in Table 7.1, our FaultTracer approach [150–152] introduced the extended call graph representation and enabled more accurate change impact analysis, in terms of both test selection and regression fault localization. In addition, we also presented the ReMT approach [155], which is inspired by traditional regression test selection to incrementally collect mutation testing results based on program differences.

7.1.2 Test Prioritization

Test prioritization reorders regression tests to detect program faults faster. Rothermel et al. [116] proposed two general strategies for test prioritization: (1) the *total* strategy which prioritizes tests based on the number of code elements they covered, (2) the *additional* strategy which prioritizes tests based on the number of additional elements they covered. Elbaum et al. [39] conducted an extensive empirical study of both total and additional strategies using various code coverage. As test coverage information may be

absent when applying test prioritization, we presented an approach to use static analysis to compute estimated code coverage for test prioritization in absence of coverage information [88,159]. Recently, we also presented proposed unified models for test prioritization which subsume the *total* and *additional* strategies as extreme cases, and also contain a spectrum of general strategies between the *total* and *additional* strategies [148]. As shown in Table 7.1, our previous work, ReMT [155], first proposed to apply test prioritization for mutation testing, but it requires mutation testing results on old versions. Our FaMT work [153] aims to present a more general test prioritization approach for mutation testing, which does not require mutation testing results on old versions. The basic intuitions for traditional test prioritization and our FaMT prioritization are similar: to reorder the tests to make regression/mutation testing faster. However, the FaMT prioritization is totally different from traditional test prioritization. The main reason is that in regression testing we do not know where the real faults are and can only use other related information (e.g., coverage information) to guide test prioritization, while in mutation testing we know the locations for all mutation faults, and can directly use fault execution information to guide prioritization.

7.1.3 Test Reduction

Test reduction executes only a representative subset of regression tests which can still satisfy all the testing requirements. Harrold et al. [50] were inspired by the fact that some *essential* tests should be selected as early as

possible because they test rarely tested requirements, and proposed a heuristic for iteratively selecting essential tests. Chen et al. [22] further found that some *redundant* tests which test only a subset of requirements tested by other tests should be reduced as early as possible, and proposed to reduce tests by iteratively applying essential test selection and redundant test reduction. Black et al. [18] considered integer linear programming models for reducing regression tests. As test reduction can lose fault detection capability, Dan et al. [47] proposed an on-demand approach which allows user to specify the acceptable loss in fault detection when reducing tests. In addition, we also empirically studied the cost-effectiveness of traditional test reduction techniques for real-world Java programs with JUnit test suites [154]. While test reduction has been proposed for decades, to our knowledge, similar ideas have not been applied to reduce the cost of mutation testing. In other words, our FaMT approach [153] is the first to apply test reduction for faster mutation testing. As Table 7.1 shows, our FaMT reduction shares the same intuition with traditional reduction: to execute only a subset of tests instead of the entire test suite. However, the detailed FaMT reduction approach is technically different from the traditional reduction techniques. Also, FaMT reduction aims to predict which mutant(s) cannot be killed precisely, while traditional reduction aims to capture most program faults.

7.2 Mutation Testing

Mutation testing [6, 32, 42, 46, 83, 122, 149, 156] is a methodology for assessing quality of test suites. Research related to mutation testing mainly contains three areas: mutation cost reduction, equivalent mutant detection, and applications of mutation testing. We describe each category in the following subsections.

7.2.1 Reducing Cost of Mutation Testing

Traditionally, there are mainly three ways to reduce the cost of mutation testing: selective mutation testing, weakened mutation testing, and accelerated mutation testing.

Selective mutation testing selects a representative subset of all mutants that can achieve similar results as the entire set of mutants. Since the first proposal of selective mutation testing by Mathur [84], a large body of research efforts has been dedicated to this area. Researchers [16, 92, 96] have experimentally investigated subsets of mutation operators to ensure that those representative sets of operators achieve almost the same results as the whole mutation operator set. Acree et al. [9] first proposed to select mutants randomly from the whole set rather than based on mutation operators. Although random mutant selection does not attract much research attention, Zhang et al. [149] recently demonstrated that operator-based selection is actually not more effective than random selection. Instead of investigating operator-based and random mutant selection separately, our *sampling mutation testing*

work [147] showed that operator-based and random mutant selection can be applied in tandem to further reduce mutation testing cost. In addition, our recent work also extensively studied selective mutation testing for concurrent code [43].

Weakened mutation testing proposes a relaxed definition of mutant killing. In traditional *strong mutation*, for program p, test t kills mutant m if and only if the final *output* of executing t on m differs from the final output of executing t on p. Howden [53] first proposed the concept of *weak mutation*, which checks whether t produces a different program internal *state* when executing m than when executing p. Later, Woodward and Halewood [137] proposed *firm mutation*, which is a spectrum of techniques between weak and strong mutation. Offutt and Lee [97] then experimentally investigated the relationships between the weak mutation and the strong mutation.

Accelerated mutation testing uses efficient ways to generate, compile, and execute mutants. DeMillo et al. [31] extended compilers to compile all mutants at once, thus reducing the cost of generating and compiling mutants. Similarly, Untch et al. [131] proposed schema-based mutation, which integrates all mutants into one meta-mutant that can be compiled by a standard compiler. Researchers have also used parallel processing [85,100] to speed up the execution of mutation testing.

Recently, we introduced the idea of test selection from the regression testing area to the mutation testing area, and proposed the ReMT technique [155], which relies on program differences and incrementally collects

mutation testing results based on old mutation testing results of a previous version. Although ReMT includes an initial test prioritization approach, it is based on program differences, and cannot be applied without mutation testing results on old versions. In contrast, our FaMT work introduces the general idea of *test prioritization* and *test reduction* to the area of mutation testing [153]. Furthermore, the FaMT techniques do not rely on program differences, and can directly apply to any programs without old mutation testing results. Both our ReMT and FaMT techniques are orthogonal to existing techniques that optimize mutation testing and can be directly combined with those techniques to further reduce the cost of mutation testing.

7.2.2 Detecting Equivalent Mutants

Equivalent mutants are mutants that are semantically identical to the original program. As equivalent mutants would impact the calculation of a test suite's quality, it is preferable to identify equivalent mutants for mutation testing. However, the problem of detecting equivalent mutants is undecidable in general. Therefore, researchers investigate approximation techniques for this problem. Offutt and Craft [94] proposed a technique to detect equivalent mutants via compiler optimization. Hierons et al. [52] used slicing to reduce the numbers of possible equivalent mutants. Schuler et al. [122] proposed to use execution information to detect equivalent mutants. While not directly targeting detection of equivalent mutants, our ReMT technique [155] may also be utilized for this purpose: ReMT determines that some tests have the same

execution on a mutant for the old and new versions and thus may reduce the cost of collecting execution information needed by equivalent mutant detection [122].

7.2.3 Applications of Mutation Testing

Mutation testing was first proposed for assessing test quality [32, 46, 83, 122]. Later, some studies have even shown that mutation testing can be more suitable than manual fault seeding in simulating real program faults for software testing experimentation [13, 36]. A number of research projects also use mutation testing to guide the automated generation of high-quality tests. DeMillo and Offutt [34] proposed constraint-based testing (CBT), which uses control-flow analysis and symbolic evaluation to generate tests each killing one mutant. Offutt et al. [95] further proposed dynamic domain reduction to address some limitations of CBT. In addition to developing dedicated test-generation techniques to kill mutants, researchers have also used existing test generation engines to kill mutants. Liu et al. [80] proposed to generate tests each killing multiple mutants using an engine based on the iterative relaxation method [45]. Fraser and Zeller [42] used search-based software testing (SBST) to generate tests that kill mutants. Zhang et al. [156] and Papadakis et al. [103] used dynamic symbolic execution (DSE) to generate tests that kill mutants. Harman et al. [49] combined SBST and DSE to generate tests that kill multiple mutants. Our ReMT approach [155] may be also utilized to reduce the cost of generating such high-quality test suites: it is possible to

incrementally generate tests for killing mutants in a way similar to augmenting existing test suites [140]. Our FaMT approach [153] may also be utilized to prioritize program paths to generate tests with higher-quality first.

A recent work by Papadakis et al. [104] is closely related to our FIFL work [158]. Their work to our knowledge is the first to utilize mutation testing to facilitate traditional fault localization. Our FIFL differs from their work in three key ways. First, their work only considers mutation changes and aims to localize faults for one specific version, whereas FIFL considers two dimensions of changes, including mutation changes and programmer edits, and aims to localize faulty edits during software evolution. Second, our approach is different: their approach uses mutation testing as a coverage criterion, whereas FIFL uses mutation testing to simulate the impact of program edits. Third, their technique is not applicable to tests with assertions, which are widely used in real-world systems. The reason is that they directly apply mutation testing on the faulty program with failed tests and the set of mutants killed by already failed tests cannot be determined. In contrast, FIFL applies to more general forms of tests because FIFL applies mutation testing on the old version where all tests pass.

Chapter 8

Conclusion

My thesis is that an *unified*, *bi-dimensional*, and *change-directed* methodology can form the basis of novel techniques and tools that can make testing and debugging significantly more effective and efficient and allow us to find more bugs at a reduced cost. My view of change is bi-dimensional: (1) we can mechanically induce changes to code or specifications to lead to higher quality test suites as originally conceived in mutation testing; (2) we can utilize manual changes made by programmers to test evolving programs more efficiently as originally conceived in regression testing. Based on this view, I proposed a set of techniques to combine mechanical changes with manual changes (i.e., combining mutation testing with regression testing) to make both regression testing and mutation testing more effective and efficient. I believe that this work can lay the foundation of more scalable and powerful techniques that are based on a synergistic application of multiple dimensions of change.

A conceptual opposite of change is *invariant*, e.g., a property or expression that is *constant* throughout a certain range of operations. I recently introduced a novel technique for precise discovery of likely program invariants based on symbolic execution and model checking [157]. I also plan to inves-

tigate the fundamental relationship between software changes and program invariants, which can be leveraged to develop more effective synergistic techniques. For example, precise program invariants can help validating program changes while program changes can help incremental invariant generation. I believe the connection between changes and invariants can further enable a suite of novel techniques for more efficient and effective software testing and analysis.

Bibliography

[1] ASM ByteCode Manipulation Framework. http://asm.ow2.org/.

[2] Eclipse JDT. http://www.eclipse.org/jdt/.

[3] Javalanche Mutation Tool. http://www.st.cs.uni-saarland.de/mutation/.

[4] MuJava Mutation Tool. http://cs.gmu.edu/~offutt/mujava/.

[5] OriginLab. http://www.originlab.com/.

[6] PIT Mutation Tool. http://pitest.org/.

[7] Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.html.

[8] R. Abreu, P. Zoeteweij, and A. Van Gemund. On the accuracy of spectrum-based fault localization. 2007.

[9] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, 1979.

[10] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE TSE*, 36(6):742–762, 2010.

[11] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press.

[12] P. Ammann and J. Offutt. *Introduction to Software Testing*. 2008.

[13] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[14] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE TSE*, pages 608–624, 2006.

[15] Barbecue Home. `http://barbecue.sourceforge.net/`.

[16] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2):113–136, 2001.

[17] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proc. ICSE*, pages 82–91, 2006.

[18] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE*, 2004.

[19] L. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proc. ICSE*, pages 86–95, 2004.

[20] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. POPL*, pages 220–233, 1980.

[21] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, 2008.

[22] T. Chen and M. Lau. A new heuristic for test suite reduction. *IST*, 40(5), 1998.

[23] O. Chesley, X. Ren, B. Ryder, and F. Tip. Crisp–A Fault Localization Tool for Java Programs. In *Proc. ICSE*, pages 775–779, 2007.

[24] P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *IEEE TSE*, 35(4):452–469, 2009.

[25] Apache Commons Home. `http://commons.apache.org/proper/commons-lang/`.

[26] Instrumented Container Classes - Predicate Coverage. `http://mir.cs.illinois.edu/coverage/`.

[27] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proc. ISSTA*, pages 207–218, 2010.

[28] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proc. ASE*, pages 433–444, 2009.

[29] M. E. Delamaro and J. C. Maldonado. Proteum–A tool for the assessment of test adequacy for C programs. In *Proc. the Conference on Performability in Computing Sys PCS 96*, pages 79–95, 1996.

[30] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/im 2.0: An integrated mutation testing environment. In *Mutation testing for the new century*, pages 91–101. Springer, 2001.

[31] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proc. COMPSAC*, pages 351–356, 1991.

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[33] R. A. DeMillo and R. J. Martin. The Mothra software testing environment user's manual. Technical report, Software Engineering Research Center, 1987.

[34] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE TSE*, 17(9):900–910, 1991.

[35] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[36] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE TSE*, pages 733–752, 2006.

[37] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[38] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

[39] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, pages 159–182, 2002.

[40] Our FaMT Project. `https://webspace.utexas.edu/~lz3548/issta13support.html`.

[41] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *JSS*, 38(3):235–253, 1997.

[42] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proc. ISSTA*, pages 147–158, 2010.

[43] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. ISSTA*, pages 224–234, 2013.

[44] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[45] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proc. FSE*, pages 231–244, 1998.

[46] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE TSE*, pages 279–290, 1977.

[47] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proc. ICSE*, pages 738–748, 2012.

[48] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. Vida: Visual interactive debugging. In *Proc. ICSE*, pages 583–586. IEEE Computer Society, 2009.

[49] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.

[50] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3):270–285, 1993.

[51] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proc. OOPSLA*, pages 312–326, 2001.

[52] R. Hierons and M. Harman. Using program slicing to assist in the detection of equivalent mutants. *STVR*, 9(4):233–262, 1999.

[53] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE TSE*, pages 371–379, 1982.

[54] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[55] Jaxen Home. `http://jaxen.codehaus.org/`.

[56] JDepend Home. `http://clarkware.com/software/JDepend.html`.

[57] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *ICSM*, pages 549–558, 2005.

[58] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *TSE*, 32(2):108–123, 2007.

[59] Y. Jia and M. Harman. Higher order mutation testing. *IST*, 51(10):1379–1393, 2009.

[60] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, 2011.

[61] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 257–266, 2009.

[62] B. Jiang, Z. Zhang, T. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proc. COMPSAC*, volume 1, pages 99–106. IEEE, 2009.

[63] Apache JMeter Home. `http://jmeter.apache.org/`.

[64] JavaSourceMetric Home. `http://sourceforge.net/projects/jsourcemetric/`.

[65] Joda Time Home. `http://joda-time.sourceforge.net/`.

[66] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE*, page 477. ACM, 2002.

[67] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE TSE*, pages 195–209, 2003.

[68] JTopas Home. `http://jtopas.sourceforge.net/jtopas/index.html`.

[69] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE TSE*, 34(2):181–196, 2008.

[70] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[71] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analyses for java software. Technical report, CSE, UNL, 2006.

[72] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE*, 32(12):971–987, 2006.

[73] E. W. Krauser, A. P. Mathur, and V. Rego. High performance software testing on SIMD machines. *IEEE TSE*, 17(5):403–423, 1991.

[74] W. B. Langdon, M. Harman, and Y. Jia. Multi objective higher order mutation testing with genetic programming. In *Proc. TAIC PART*, pages 21–29, 2009.

[75] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. ICSE*, pages 3–13, 2012.

[76] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritisation. *TSE*, 33(4):225–237, 2007.

[77] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. PLDI*, pages 15–26. ACM, 2005.

[78] S. Lin. Computer solutions of the travelling salesman problem. *Bell System Technical Journal*, 44(5):2245–2269, 1965.

[79] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[80] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun. An approach to test data generation for killing multiple mutants. In *Proc. ICSM*, pages 113–122, 2006.

[81] R. Lowry. *Concepts and applications of inferential statistics*. R. Lowry, 1998.

[82] Y. S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *STVR*, 15(2):97–133, 2005.

[83] Y. S. Ma, J. Offutt, and Y. R. Kwon. Mujava: a mutation system for java. In *Proc. ICSE*, pages 827–830, 2006.

[84] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proc. COMPSAC*, pages 604–605, 1991.

[85] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report, Purdue University, 1988.

[86] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proc. OOPSLA*, pages 683–702, 2012.

[87] T. McCabe. A complexity measure. *TSE*, (4):308–320, 1976.

[88] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38(6):1258–1275, 2012.

[89] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[90] Apache Mime4J Home. `http://james.apache.org/mime4j/`.

[91] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proc. ICSE*, pages 332–341, 2013.

[92] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.

[93] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, May 2002.

[94] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *STVR*, 4(3):131–154, 1994.

[95] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, 1999.

[96] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM TOSEM*, 5(2):99–118, 1996.

[97] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE TSE*, 20(5):337–344, 1994.

[98] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *STVR*, 7(3):165–192, 1997.

[99] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. International Conference on Testing Computer Software*, 1995.

[100] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using MIMD computer. In *Proc. International Conference on Parallel Processing*, pages 257–266, 1992.

[101] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc. ICSE*, 1993.

[102] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. FSE*, pages 241–251, 2004.

[103] M. Papadakis, N. Malevris, and M. Kallia. Towards automating the generation of mutation tests. In *Proc. AST*, pages 111–118, 2010.

[104] M. Papadakis and Y. L. Traon. Using mutants to locate unknown faults. In *Proc. ICST Workshop on Mutation Analysis*, pages 691–700, 2012.

[105] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proc. ISSTA*, pages 199–209, 2011.

[106] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proc. PLDI*, pages 504–515, 2011.

[107] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proc. ISSTA*, pages 191–201, 2013.

[108] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.

[109] X. Ren and B. Ryder. Heuristic Ranking of Java Program Edits for Fault Localization. In *Proc. ISSTA*, pages 239–249, 2007.

[110] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proc. OOPSLA*, 2004.

[111] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE*, pages 30–39, 2003.

[112] T. Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[113] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *TOSEM*, 13(3):227–331, 2004.

[114] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[115] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *STVR*, 12(4):219–249, 2002.

[116] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *TSE*, 27(10), 2001.

[117] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proc. ICSM*, pages 179–188, 1999.

[118] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. PASTE*, pages 46–53, 2001.

[119] Our Sampling Mutation Project. `https://webspace.utexas.edu/~lz3548/ase13support.html`.

[120] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proc. ICSE*, pages 56–66, 2009.

[121] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. ISSTA*, pages 69–80, 2009.

[122] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. FSE*, pages 297–298, 2009.

[123] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. FSE*, pages 263–272, 2005.

[124] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[125] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Proc. FASE*, pages 262–277, 2011.

[126] M. Staats, G. Gay, and M. P. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proc. ICSE*, pages 870–880, 2012.

[127] M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *Proc. FASE*, pages 409–424. 2012.

[128] M. Stoerzer, B. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proc. FSE*, pages 57–68, 2006.

[129] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.

[130] Time and Money Home. `http://timeandmoney.sourceforge.net/`.

[131] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proc. ISSTA*, pages 139–148, 1993.

[132] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. ICSE*, pages 364–374, 2009.

[133] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[134] L. J. Williams and H. Abdi. Fisher's least significance difference (LSD) test. In *Encyclopedia of Research Design*, pages 491–494. Thousand Oaks, 2010.

[135] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 230–238, 1997.

[136] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *JSS*, 31(3):185–196, 1995.

[137] M. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proc. the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, 1988.

[138] Santuario Home. `http://santuario.apache.org/`.

[139] XStream Home. `http://xstream.codehaus.org/`.

[140] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proc. FSE*, pages 257–266, 2010.

[141] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *Proc. ICSM*, pages 115–124, 2009.

[142] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *STVR*, 22(2), 2012.

[143] S. Yoo, M. Harman, and S. Ur. Measuring and improving latency to avoid test suite wear out. In *ICSTW*, pages 101–110, 2009.

[144] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proc. ICSE*, pages 201–210. ACM, 2008.

[145] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Prc. FSE*, pages 253–267, 1999.

[146] A. Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[147] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Proc. ASE*, pages 92–102, 2013.

[148] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proc. ICSE*, page to appear, 2013.

[149] L. Zhang, S. S. Hou, J. J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.

[150] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proc. ICSM*, pages 23–32. IEEE, 2011.

[151] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. 2012.

[152] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: a spectrum-based approach to localizing failure-inducing program edits. *JSME*, 25(12):1357–1383, 2013.

[153] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*, pages 235–245, 2013.

[154] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *Proc. ISSRE*, pages 170–179, 2011.

[155] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341. ACM, 2012.

[156] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.

[157] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proc. ISSTA*, page to appear, 2014.

[158] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proc. OOPSLA*, page to appear, 2013.

[159] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing junit test cases in absence of coverage information. In *Proc. ICSM*, pages 19–28, 2009.

[160] S. Zhang. Practical semantic test simplification. In *Proc. ICSE*, pages 1173–1176, 2013.

[161] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proc. ASE*, pages 63–72. IEEE, 2011.

[162] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proc. ICSE*, pages 272–281. ACM, 2006.

[163] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proc. of PLDI*, pages 169–180, 2006.

[164] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4), Dec. 1997.

[165] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proc. ICSE*, pages 1074–1083, 2012.