Efficient Mutation Analysis for Industrial Software
(産業用ソフトウェアに対する効率的なミューテーション解析)

by

Susumu Tokumoto
徳本晋

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on June 5, 2020
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Shinichi Honiden　本位田真一
Professor of Computer Science

ABSTRACT

Industrial software is developed and maintained by many people with different experience and skills within a limited time and cost. It is known that the lack of experience and skills of developers greatly affects the number of defects in the software. In addition, because of the rapid turnover of developers in companies, there are many legacy systems that do not have quality assurance and cannot be identified by whoever created them. Due to the lack of experience and skills of the developers, they may make changes to the parts of the software that are not the root cause of the bugs, and as a result, if the bugs are not fully fixed, multiple defects may occur. Similarly, if the cause of a bug in a legacy system is located in a place that cannot be changed, it must be dealt with by making changes in a place that can be changed, but again, if the bug is not fully fixed, multiple intertwined defects will occur. If these defects are not detected before the release of the software, there is a possibility that significant social damage will occur. One of the best known techniques for detecting complex defects is mutation analysis. It is a method to measure the ability of a given test to detect bugs that are artificially embedded by mutating the elements of the program (mutation). Mutation analysis is not only used to measure the bug detectability of a test; it also has a wide range of applications, such as high-precision fault localization and automated program repair. Higher order mutation analysis, which mutates multiple locations simultaneously, is a technique that has greater potential for detecting complex defects than ordinary mutation analysis. However, higher order mutation analysis generates a large number of mutants (mutated programs), and all of them need to be compiled and tested, which is a problem that results in a very long execution time. Therefore, higher order mutation analysis has not been widely used in industry, despite the fact that strong applications have been devised. We propose, implement, and evaluate a high-speed higher order mutation analysis method and a mutant optimization technique to improve execution costs to enable the application of higher order mutation analysis to industrial software.

For the high-speed higher order mutation analysis method, we use four techniques to achieve improved execution time for mutation analysis: metamutation, virtual machines (VMs) for mutation, runtime mutation application, and high-order stream split execution. First, to avoid losing the information in the source code to the intermediate representation by, e.g., compile-time optimization, the program elements are replaced by a function called the metamutation function, which indicates the mutation position and type before compilation. This enables the mutation of intermediate expressions to match the mutation to the source code. Second, by running it on a VM, we reduce the overhead by starting a process one time instead of running it per test. Third, rather than rewriting the source code for each mutant, we use the mutant information for each execution state to translate the instructions into mutated ones when executing the intermediate expression. This technique can shorten the compilation time because it only needs to be compiled once. Fourth, while preserving the execution state on the VM, at the time of execution of each instruction, it branches into a state that executes the original instruction and a state that executes the mutated instruction. This makes it possible to shorten the test execution time. We conducted comparative experiments, which indicate that our method is significantly superior to an existing tool, an existing technique (mutation schema generation), and no-split-stream execution in higher order mutation.

For the mutant optimization, we analyze the limits of the reduction of mutants without loss of reliability. Existing methods remain challenged in terms of excessive mutant reduction and errors in the mutation score after reduction. The results of evaluation using open source software (OSS) show that the greedy mutant selection method reduces the execution time by approximately 40%, although the reduction is inferior to that of the existing method. To evaluate the impact of the reduction of excess mutants, we measured the mutation score for the test in which the bug detectability was artificially reduced, and the discrepancy in the mutation score of the proposed method was less than that of the existing method.

Leveraging the high-speed mutation analysis foundation, we improve the efficiency of the fault localization technique as an application of mutation analysis. Fault localization is a technique to reduce the cost of debugging by ranking candidate fault causes based on test results and test execution information. Among the several fault localiza-

tion techniques, mutation-based fault localization (MBFL) can localize faults with high accuracy but has the problem of high execution cost. Meanwhile, in mutation analysis, it is known that the statement deletion mutation operator has less bias in mutation points and is as effective as using all mutation operators even when used alone. Therefore, we implemented MBFL using only the statement deletion mutation operator (SDL-MBFL) and evaluated the localization of the software used in the actual product and nine actual faults. As a result of the evaluation, SDL-MBFL found more faults than existing methods in the higher ranks of 100 or more.

The overall evaluation of this study in terms of the application of mutation analysis to bug detectability measurement is that it contributes to methods to improve speed and accuracy, and has an advantage over state-of-the-art techniques, especially in higher order mutation. In terms of the application of mutation analysis to fault localization, this work is superior to the state-of-the-art in granularity and reliability. These contributions will greatly reduce the cost of performing higher order mutation and enable the detection of complex defects in industrial software.

論文要旨

産業用ソフトウェアは限られた時間とコストの中で経験・スキルの差のある多人数によって開発・保守が行われる．開発者の経験・スキルの不足はソフトウェア内の欠陥数に大きく影響することがわかっている．また企業では開発者の入れ替わりが激しいため，誰が作ったかわからなく品質の保証がないレガシーシステムが多く存在する．開発者の経験・スキルの不足のために，バグの根本原因ではない箇所に変更が入り，結果的にバグが十分に修正しきれなかった場合，複数個所が交絡した欠陥が生じてしまう．同様に，レガシーシステムにおいて変更が不可能な箇所にバグの原因がある場合，変更可能な箇所での変更によって対処しなければならないが，やはりバグが十分に修正しきれなければ，複数個所が交絡した欠陥が生じてしまう．

複雑な欠陥の検出のための技術としてミューテーション解析が知られている．ミューテーション解析はプログラムの要素を変異（ミューテーション）させて人為的にバグを埋め込むことで，用意されたテストがどれだけその埋め込まれたバグを検知できる能力を持っているか測定する手法である．また，高精度の欠陥局所化，プログラム自動修復などの幅広い応用も考えられている．高次ミューテーション解析は複数個所を同時に変異させるもので，通常のミューテーション解析よりも複雑な欠陥を検出するのに大きな可能性がある技術である．高次ミューテーション解析は大量のミュータント（変異させたプログラム）を生成し，それらをすべてコンパイルし，テストをする必要があるため，実行時間が非常に長くなる問題がある．そのため，高次ミューテーション解析は強力な応用分野が考案されているにも関わらず，産業界では広く使わていない状況にある．本研究では，産業用ソフトウェアへの高次ミューテーション解析の適用を可能にすべく，実行コスト改善のために高速実行手法のアプローチとミュータント最適化のアプローチを提案，実装，評価した．

高速実行手法のアプローチでは，メタミューテーション，ミューテーション用仮想機械（VM），実行時ミューテーション適用，高次ストリーム分割実行の 4 つの技術を用いて，ミューテーション解析の実行時間の改善を達成した．まずソースコード上の情報がコンパイル時の最適化などによって中間表現では失われないよう，コンパイル前にプログラム要素をメタミューテーション関数と呼ばれるミューテーション位置と種類を表す関数に置き換えることで，中間表現に対してもソースコードへのミューテーションと一致するミューテーションを可能にした．次に，VM 上で実行することで．プロセスをテストごとに起動していたのを 1 回のプロセス起動で済むようにし，これによりプロセス起動によるオーバヘッドを削減した．また，ソースコードを書き換えず，中間表現の命令実行時に各実行状態のミュータント情報から命令を読み替えて実行する．これにより 1 回のコンパイルで十分となるためコンパイル時間の短縮が可能になる．さらに，VM 上で実行状態を保存しながら実行し，各命令の実行時にミューテーションした命令を実行する状態へ分岐する．これによりテスト実行時間の短縮が可能になる．これらの手法を評価するため，比較実験として我々の手法と既存ツール，既存手法と比較を行い，我々の手法が有為に優れていることを確認した．

ミュータント最適化のアプローチでは，信頼性を損なわないミュータントの削減量の限界を解析した．既存手法は過剰なミュータント削減と削減後のミューテーションスコアの誤差という点で課題を残している．提案手法ではミューテーション解析中に引き起こされるエラーの種類を網羅するようなミュータントを選択することと，選択されたミュータン

トに重み付けをすることで過剰なミュータント削減を防ぐ高信頼なミューテーションスコアの計測方法を提案する．OSS を用いて評価した結果，既存手法よりも削減量は劣るものの，貪欲法によるミュータント選択方法により約 40%の実行時間の削減効果が得られた．また，過剰なミュータントの削減の影響を評価するために，バグ検出力を人為的に落としたテストに対してミューテーションスコアを計測したところ，既存手法に比べ提案手法のミューテーションスコアの誤差は少なくなった．

さらにミューテーション解析の応用として，その高速ミューテーション解析基盤を活用し欠陥局所化技術の効率化を行った．欠陥局所化技術はテスト結果やテスト実行情報などから欠陥の原因個所の候補を順位付けすることでデバッグの作業コストを削減するための技術である．いくつかの欠陥局所化技術の中で，ミューテーション解析に基づく欠陥局所化技術 (MBFL) は高い精度で欠陥を局所化できるが，実行コストが高い問題がある．一方，ミューテーション解析において，命令削除ミューテーションオペレータはミューテーション箇所の偏りが少なく，単独での利用でも全てのミューテーションオペレータを使った場合と同等の効果があることが知られている．そこで命令削除ミューテーションオペレータのみを用いた MBFL(SDL-MBFL) を実装し，実際の製品で使われているソフトウェアと実際に起こった 9 件の欠陥に対して局所化の評価を行った．評価の結果として，SDL-MBFL は 100 位以上の高い順位において既存手法より多くの欠陥箇所を挙げられた．

ミューテーション解析のバグ検出力測定への応用の観点での本研究の全体の評価としては，速度や精度を改善する手法について貢献があり，特に最先端技術に対して高次ミューテーションで優位性がある．また，ミューテーション解析の欠陥局所化への応用の観点では，本研究は最先端技術と比べ粒度と信頼度において優れている．このような貢献により高次ミューテーションの実行コストの大きな削減を実現し，産業用ソフトウェアにおける複雑な欠陥の検出を可能にするものと考える．

Acknowledgements

First, I would like to express my deepest gratitude to my supervisor, Professor Shinichi Honiden, for his invaluable advice, persistent support, and strong encouragement, which helped not only my study but also my lifestyle and ways of thinking as a researcher. In addition, he patiently mentored me during and beyond my doctoral course, without which I would not have been able to continue the challenge of this study.

I would also like to sincerely thank my PhD committee members—Professor Masami Hagiya as chair, Professor Naoki Kobayashi, Professor Yusuke Miyao, Professor Akihiko Takano, and Associate Professor Shinpei Kato—for their insightful, constructive, and informative feedback, which refined the thesis.

My heartfelt gratitude extends to Dr. Hiroaki Yoshida at Fujitsu Laboratories of America for guiding me in implementation and giving lots of technical advice. Without his dedicated help, this study would not have been possible. I would also like to offer my special thanks to Dr. Mukul Prasad at Fujitsu Laboratories of America. His feedback, based on his extensive experience in software engineering research, has contributed significantly to the quality of my study.

Additionally, I am deeply indebted to faculties and ex-faculties in the Honiden laboratory, especially Associate Professor Kazunori Sakamoto who provided me illuminating and important discussions, and Associate Professor Fuyuki Ishikawa, Associate Professor Kenji Tei, Professor Yoshinori Tanabe, Assistant Professor Ryuichi Takahashi, Dr. Yuta Maezawa, and Assistant Professor Soramichi Akiyama who provided me essential and valuable feedback.

Moreover, special thanks also go to all members and ex-members of the Honiden laboratory, whose comments and suggestions were an enormous help to me. They also made my life at National Institute of Informatics enjoyable. I would like to list the members, Dr. Tsutomu Kobayashi, Kohsuke Yatoh, Kazuya Aizawa, Dr. Ryo Shimizu, Fernando Tarin Morales, Takayuki Suzuki, Shun Lee, Junto Nakaoka, Natsumi Asahara, Yuta Tokitake, Miki Yagita, Masaki Katae, Moeka Tanabe, Yasuo Tsurugai, Yasuhiro Sezaki, Katsuhiko Ikeshita, Takaya Saeki, Shinnosuke Saruwatari, Daichi Morita, Tomoya Katagi, Paul Harvey, Aurélien Vialon, Kazuyuki Honda, Takahiro Sugiura, Keita Tsukamoto, Yetian Mao, Koki Kato, and Chihiro Iida.

I would like to sincerely thank the directors, managers, and fellows at Fujitsu Laboratories Ltd., especially Dr. Rieko Yamamoto and Tadahiro Uehara for allowing me to have this opportunity, and Kenichi Abiru, Hidetoshi Kurihara, Dr. Shinji Kikuchi, Kuniharu Takayama, Kazuki Munakata, Isao Nanba and Takeshi Yasuie for their generous support and encouragement. My sincere thanks also go to my co-workers at Fujitsu Laboratories Ltd. who showed understanding in my PhD study, including Atsuji Sekiguchi, Koki Kato, Yuuji Hotta, Satoshi Munakata, Masaru Ueno, Katsuhisa Nakazato, Dr. Hideo Tanida, Yusuke Nemoto, Dr. Keisuke Hotta, Dr. Kunihiro Noda, Sho Maeda, Haruki Yokoyama, Takumi Akazaki, and Dr. Yusuke Kimura. I would also like to genuinely thank the

executives of Fujitsu Laboratories Ltd. for their financial support.

Furthermore, I would like to tender my cordial thanks to Professor Hiroshi Imai who was my master's supervisor for his warm encouragement. Also, the talented members of the Imai laboratory always stimulated my desire to challenge myself in my PhD.

Finally and most importantly, I am sincerely grateful to my family for their selfless support. My wife Yuki, my daughter Fuuka, and my son Souichi were always there to encourage me.

Contents

1 Introduction **1**
1.1 The State of Industrial Software 1
1.2 Background of Mutation Analysis 1
1.3 Needs of Measuring Software Coverage in Industry 2
1.4 Problem and Motivation of Mutation Analysis 2
1.5 Application of Mutation Analysis: Mutation-based Fault Localization 3
1.6 Approach Overview 4
1.7 Contributions 4
1.8 Organization 5

2 Background on Mutation Analysis **6**
2.1 Terminology of Software Problems 6
2.2 Software Testing 7
2.3 Mutation Analysis for Assessing Test Quality 9
2.4 Fundamental Hypotheses 11
2.5 Process of Mutation Analysis 11
2.6 Computational Cost of Mutation Analysis 12

3 A Systematic Literature Review of Code Coverage Measurement in Industrial Testing **14**
3.1 Overview 14
3.2 Research Method 14
3.2.1 Goal and Research Questions 15
3.2.2 Research Process 15
3.3 Results 19
3.3.1 RQ1: Which programming languages of SUT are popular for coverage measurement? 19
3.3.2 RQ2: What types of coverage criteria are used? 19
3.3.3 RQ3: For what purpose is coverage used? 19
3.3.4 RQ4: What effects have resulted from the use of coverage? 20
3.3.5 RQ5: What quality characteristics are required in coverage measurement tools? 20
3.4 Discussion 23
3.4.1 Context Type of Quality Characteristics 23
3.4.2 Needs for Coverage Measurement in Industry 24
3.5 Summary 24

4 Virtual Machine for Mutation Analysis **26**
4.1 Overview 26
4.2 Preliminary 27
4.2.1 Mutant Schemata Generation 27

4.2.2 Bitcode Translation . . . 28
4.2.3 Split-stream Execution . . . 29
4.2.4 Higher Order Mutation . . . 30
4.3 Techniques . . . 30
4.3.1 Metamutation . . . 30
4.3.2 Mutation on Virtual Machine . . . 32
4.3.3 Higher Order Split-stream Execution . . . 32
4.3.4 Online Adaptation Technique . . . 34
4.4 Design of MuVM . . . 34
4.4.1 Overall Structure and Behavior . . . 34
4.4.2 Complications . . . 36
4.4.3 Mutation Score Calculation . . . 37
4.5 Evaluation . . . 38
4.5.1 Competitive Tools . . . 39
4.5.2 Subject Programs . . . 39
4.5.3 Experimental Procedure . . . 39
4.5.4 Hypothesis . . . 40
4.5.5 Results and Discussion . . . 40
4.5.6 Threats to Validity . . . 44
4.6 Summary . . . 44

5 Statement Deletion Mutation-based Fault Localization 46
5.1 Overview . . . 46
5.2 Preliminary . . . 46
5.2.1 Statement Deletion Mutation . . . 47
5.2.2 Spectrum-based Fault Localization . . . 47
5.2.3 Mutation-based Fault Localization . . . 48
5.2.4 Statement Deletion Mutation-based Fault Localization . . . 49
5.2.5 MBFL and SBFL Hybrid Approach . . . 50
5.3 Evaluation Setup . . . 51
5.3.1 Research Questions . . . 51
5.3.2 Tool . . . 52
5.3.3 Evaluation Subjects . . . 52
5.3.4 Evaluation metrics . . . 53
5.4 Evaluation Results . . . 54
5.4.1 RQ1: How long does each mutation analysis run? . . . 55
5.4.2 RQ2: What is a good formula for calculating the suspiciousness of SDL-MBFL? . . . 55
5.4.3 RQ3: Does SDL-MBFL rank high in faults compared to other fault localization methods? . . . 55
5.4.4 RQ4: Does the hybrid method of SDL-MBFL and SBFL rank high in faults? . . . 55
5.5 Discussion . . . 59
5.5.1 Practical cost-effectiveness . . . 59
5.5.2 Characteristics of the faults . . . 59
5.5.3 How to choose a mutation operator . . . 61
5.6 Summary . . . 61

6 Error-Oriented Mutant Reduction and Mutant Weighting for Reliable Mutation Analysis **63**
6.1 Overview . . . 63
6.2 Preliminary . . . 64
6.3 Motivating Example . . . 65
6.4 Proposed Method . . . 66
6.4.1 Definitions . . . 67
6.4.2 Mutant Set Minimization Algorithm . . . 68
6.4.3 Mutant Weighting . . . 69
6.4.4 Example of Mutant Set Minimization . . . 70
6.4.5 Example of Mutant Weighting . . . 70
6.5 Evaluation . . . 71
6.5.1 Research Questions . . . 71
6.5.2 Evaluation Method . . . 71
6.5.3 Subject of Evaluation . . . 72
6.5.4 Evaluation Results . . . 73
6.6 Discussion . . . 77
6.6.1 Ratio of Assertion Fixes to Test Code Fix Commits . . . 77
6.6.2 Execution Time Optimization . . . 77
6.6.3 Reducing Mutation Score Discrepancy . . . 78
6.6.4 Reduction Per Mutation Operator . . . 78
6.7 Summary . . . 79

7 Related Work **82**
7.1 Speeding Up Mutation Analysis . . . 82
7.2 Mutants Optimization . . . 82
7.3 Mutation-based Fault Localization . . . 83
7.4 Industrial Case Studies of Mutation Analysis . . . 84
7.5 Evaluation of Debugging Techniques for Industrial Software . . . 84
7.6 Applications of Mutation Analysis . . . 85
7.7 Tools for Mutation Analysis . . . 85
7.8 Data flow Analysis for Testing and Debugging . . . 87

8 Conclusion **90**
8.1 Summary . . . 90
8.2 Overall Evaluation . . . 91
8.2.1 Overall Evaluation as Coverage Measurement Technique . . . 91
8.2.2 Overall Evaluation as Fault Localization Technique . . . 92
8.2.3 Overall Evaluation with Future Prospects . . . 94
8.3 Future work . . . 94
8.3.1 Further Improvement in Performance . . . 94
8.3.2 Other Practical Issues . . . 94
8.3.3 Towards Further Industrial Adoption . . . 95

A Detailed Proof that the Ratio of Computational Cost in k-th Order Split-stream Execution is $k+1$ **114**

List of Figures

1.1 Overview of our approaches 4

2.1 Terminology in Program failure 7
2.2 Test suites, test cases, test data and test oracle 8
2.3 A process of mutation testing for incremental improvement of a test suite 12
2.4 Computational cost of mutation analysis 13

3.1 Research process used to conduct this study 14
3.2 Annual trend of publications including industrial coverage measurement 16
3.3 Number of publications by programming languages 19
3.4 Number of publications by coverage criteria 20
3.5 Number of publications by purpose 21
3.6 Number of publications by effect 21
3.7 Number of publications by quality characteristics 22
3.8 Number of quality characteristics in publications mentioned 22

4.1 Mutation schemata generation 27
4.2 Bitcode translation 28
4.3 Mutants omission by optimizer 29
4.4 Split-stream execution 30
4.5 Overview of MuVM approach 31
4.6 Example of metamutation 31
4.7 Higher order split-stream execution 32
4.8 HOSSE theoretical model 33
4.9 State Transition in Offline and Online adaptation Technique 34
4.10 Structural design of MuVM 35
4.11 Metamutation for Structural Mutation 37
4.12 Decision tree for simulating infeasible HOM 38

5.1 Statement deletion mutation 47
5.2 Spectrum-based Fault Localization 50
5.3 Statement Deletion Mutation-based Fault Localization 51
5.4 Overview of system re-engineering project 52
5.5 $E_{inspect}@n$ in each of the SDL-MBFL suspiciousness calculation formulas 56
5.6 $E_{inspect}@n$ for each fault localization technique 56
5.7 $E_{inspect}@n$ for each hybrid fault localization technique 57

6.1 Ratio of modified assertions in commits of test code modification . 73
6.2 Distribution of increased/decreased assertions in commits of test code modification 74
6.3 Distribution of execution time of mutation analysis 74

6.4 Absolute values of mutation score difference in reduction of assertions 75
6.5 Absolute values of mutation score error in reduction of test code's statements in jsoup 76
6.6 Number of mutants per mutation operator 76
6.7 Reduction rate of mutants per mutation operator 77

7.1 Phanta: A Test Code Quality Measurement Tool 86

List of Tables

2.1 A comparison of mutation operators for Fortran [112], C [3], and Java (PIT) [44] . . . 10

3.1 Search keywords and number of publications . . . 16
3.2 All relevant publications in this study and their coverage data . . . 17
3.3 All relevant publications in this study and their coverage data (cont.) 18
3.4 List of publications related to coverage measurement quality . . . 23

4.1 Fundamental data of subject programs . . . 39
4.2 A Total Number of Invoked Mutants . . . 40
4.3 Execution Time . . . 41
4.4 The Results of t-test . . . 42

5.1 Overview of subject program . . . 53
5.2 Subject faults . . . 53
5.3 Example of suspiciousness ranking for calculating $E_{inspect}$ . . . 54
5.4 Results of mutation analysis . . . 55
5.5 Results of fault localization (rank and average EXAM by $E_{inspect}$) 58

6.1 Example of mutants and test . . . 64
6.2 Example of mutants and errors . . . 70
6.3 Mutation operators used for the evaluation . . . 72
6.4 Repositories used for evaluation and their LOC . . . 80
6.5 Numbers of test cases and assertions in jsoup, zt-zip, and jInstagram 81
6.6 Execution time of mutation analysis . . . 81
6.7 Number of killed mutants in reduction of assertions . . . 81
6.8 Number of killed mutants in reduction of test code's statements in jsoup . . . 81

7.1 A comparison of Mutation analysis tools for C/C++ (OO:Object-oriented mutation operators, BM:Bitcode-level mutation, MSG:Mutant schemata generation, HOM:Higher order mutation, SSE:Split-stream execution) . . . 86
7.2 A comparison of mutation analysis tools for Java (OO:Object-oriented mutation operators, BT:Bytecode translation, MSG:Mutant schemata generation, HOM:Higher order mutation, Concurrency:Concurrency-related mutation operators) . . . 87

Citations to Printed Publications

Parts of this thesis have appeared in the following publications. The circled numbers mean main publications for this thesis.

Journals

①. Susumu Tokumoto and Shinichi Honiden. “Error-Oriented Mutant Reduction and Mutant Weighting for Reliable Mutation Testing”. In: *IPSJ Journal* 61.4 (Apr. 2020), pp. 945–956

②. Susumu Tokumoto and Shinichi Honiden. “Evaluating Statement Deletion Mutation-based Fault Localization in Industrial Software”. In: *IPSJ Journal* 61.10 (Oct. 2020). (in press)

Proceedings

1. Susumu Tokumoto, Kazunori Sakamoto, Kiyofumi Shimojo, Tadahiro Uehara, and Hironori Washizaki. “Semi-automatic Incompatibility Localization for Re-engineered Industrial Software”. In: *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on.* (Industry Track). IEEE. 2014, pp. 91–94

②. Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto, and Shinichi Honiden. “MuVM: Higher order mutation analysis virtual machine for C”. in: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST).* (Research Track). IEEE. 2016, pp. 320–329

3. 徳本 晋 and 本位田 真一. “ミュータント削減手法の高信頼化に向けて”. In: ソフトウェアエンジニアリングシンポジウム *2019* 論文集. 情報処理学会. 2019, pp. 106–115

4. Susumu Tokumoto and Kuniharu Takayama. “PHANTA: Diversified Test Code Quality Measurement for Modern Software Development”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* (Industry Showcase). IEEE. 2019, pp. 1206–1207

Chapter 1

Introduction

1.1 The State of Industrial Software

Industrial software is developed and maintained within a limited time and cost by multiple people and teams with varying experience and skills. Differences in the experience and skills of developers have a significant impact on the defects in the software produced. A survey of 4067 projects in 31 companies by IPA [212] shows that those with better testing skills tend to have lower defect density than those with lower skills. Also, a study in commercial projects by Tsunoda et al. [191] found that experienced developers introduced fewer defects, regardless of the work difficulty. Compared to academia and voluntary communities, software produced in industry is more likely to have defects due to variations in the experience and skills of the developers, as many people are involved in the process under time and cost constraints.

Today, software is increasingly being used in various systems in the industry. It is also applied to mission-critical domains. This trend is reflected in the amount of economic loss due to software bugs. Economic losses due to software bugs were found to be $59.5 billion per year in a 2003 [173] and $312 billion per year in 2013 [27]. Thus, in just 10 years, the impact of software quality and reliability on the economy increased more than 5-fold. According to a survey by the Information-technology Promotion Agency [125], the number of reported information system failures has risen to an unprecedented level in recent years—48 in 2017, 66 in 2018, and 122 in 2019. Furthermore, software developers spend 49% of their development time detecting and fixing bugs [29]. These trends indicate that technologies are needed to support the test and debug process.

1.2 Background of Mutation Analysis

Although there are a number of advanced technologies for testing and debugging, they are not yet sufficient in terms of performance and functionality to be widely used in industry. To be used in many fields and to contribute to the enhancement of software quality and reliability, further advancements are still necessary. Mutation analysis, which is a testing technique for improving the fault detection capability of test suites, is expected to provide a powerful foundation for this.

Mutation analysis embeds an artificial fault into a program, by changing one program element (a mutated program is called a mutant), examines how well a given test suite can detect the fault, and repeats this until all mutants are tested. Mutation analysis is not only used for the evaluation of test suites but also has a variety of other technical applications. For example, there is a technique for automating the selection of test oracles for test design. By applying mutation

analysis to this, i.e., measuring the impact of the embedded fault on the oracle, it is possible to select more effective oracles for the fault.

1.3 Needs of Measuring Software Coverage in Industry

The most basic function of mutation analysis is coverage measurement. Knowing what quality characteristics are required for coverage measurement in industrial software will guide us in improving our testing and debugging techniques. In this thesis, we conduct a systematic literature review on coverage measurement for industrial software in order to understand what the quality of coverage measurement essentially means in industrial software development. From 151 automatically collected publications on code coverage measurement in industry, we manually extracted 62 publications and examined the coverage measurement activities described in them, their purpose, effectiveness, and required quality characteristics. The most common purpose of the coverage measurement is test evaluation, and the most common effect was defect detection, excluding evaluation in experiments. The required quality characteristics for coverage measurement include usability, speed, accuracy, scalability, memory consumption, and reliability. In the context of mutation analysis, there are some issues, especially in terms of speed and associated accuracy, that need to be addressed.

1.4 Problem and Motivation of Mutation Analysis

Running mutation analysis is much more computationally intensive than conventional testing. This is because each program element must be mutated, compiled, and test run. The time-consuming nature of the analysis has been a hurdle to its application to industrial software development. Therefore, reducing the cost of running a mutation analysis is the key to its widespread use.

Offutt and Untch [158] split cost reduction techniques for mutation analysis into three categories: *do fewer*, *do smarter*, and *do faster*. The "do fewer" approaches attempt to run fewer mutants without incurring unacceptable information loss. The "do smarter" approaches seek to distribute the computational expense over several machines or amortize the expense over several executions by retaining state information among the runs. The "do faster" approaches focus on generating and running mutants as efficiently as possible.

Existing cost reduction techniques for mutation analysis include Mutant Schemata Generation (MSG) and Bitcode Translation (do faster), Split-stream Execution (do smarter), and selective mutation and test case-based mutant optimization (do fewer).

The challenge of "do faster" approaches is that bitcode translation loses mutation locations during compile-time optimization while MSG incurs the overhead of process launch. The challenge of "do smarter" approaches is that Split-stream Execution (SSE) is applicable only for interpreter-based execution methods because naïve compiler-based methods cannot branch the execution stream. The challenge of "do fewer" approaches is that the test case-based method has too coarse a granularity of optimization, which in some cases leads to an over-reduction of mutants.

In this thesis, we propose a high-speed execution method for mutation analysis as a "do faster" and "do smarter" approach and a mutant optimization technique as a "do fewer" approach.

Our high-speed execution method consists of four techniques: *metamutation*, *mutation on a virtual machine*, *higher order split-stream execution*, and *online*

adaptation. *Metamutation* is a technique for retaining mutation information during bitcode translation; it replaces target program elements in the source code with corresponding metamutation functions, which can return all possible mutation results at runtime. *Mutation on a virtual machine* reduces compilation and process-invocation costs by processing metamutated intermediate code on our original virtual machine, which can interpret metamutation functions. *Higher order split-stream execution* implements an efficient execution method for higher order mutation, which create mutants by the insertion of two or more faults, by branching an execution state into mutated states and a non-mutated state, at the point where a metamutation function is invoked on a VM. *Online adaptation* reduces the number of mutant applications by dynamically executing the feasible mutants.

For mutant optimization, we present a technique for determining mutant redundancy by the type of errors caused by mutation. We then introduce a model that prunes away mutants with overlapping types of errors by recognizing them as redundant. In addition, we propose a method to quantify the impact of reduced mutants and to score the remaining mutants. As a result, we are able to reduce the discrepancy between the mutation score after mutant reduction and that of the original mutant set.

1.5 Application of Mutation Analysis: Mutation-based Fault Localization

The most promising example of an application of mutation analysis in industry is fault localization, which is a debugging technique that identifies locations in the code that are responsible for test failures.

When developers become aware of the existence of a fault due to a test failure, they attempt to remove the fault. However, since programs are usually large and complex, it is challenging for developers to understand the entire structure and behavior of the program. Therefore, diagnosing the cause of tests failures is a skillful and time-consuming task. Specifically, the developers' work of diagnosing the cause of the defect entails first understanding the overall structure of the program, comparing the passing and failing tests, narrowing down the lines that affect the failure, and then identifying the cause of the failure. Fault localization reduces the burden on developers by automating those tasks.

Spectrum-based fault localization (SBFL) is the most widely used method for exploiting information from execution paths of both passed and failed tests. Mutation-based fault localization (MBFL) has been proposed as a new way to develop it. SBFL treats statements that execute more failed test cases as more faulty, and conversely, statements that execute more passed test cases as less faulty. Building on this intuition, MBFL takes a mutation that affects the output of a failed test case to be more faulty and a mutation that affects the output of a successful test case to be less faulty.

Similar to mutation analysis, the long execution time of MBFL is one of the barriers to its practical use. While SBFL can obtain the relevant fault localization information by running a test set just once, MBFL must run the test set once for *each* mutant, for fault location estimation. Through this process, however, MBFL can provide a wealth of clues about the impact of each program element on the test results, i.e., the cause of the defect.

We implement MBFL, enabling us to predict fault locations more accurately in a shorter time, by using a statement deletion mutation operator (SDL), in

addition to a fast mutation analysis tool.

1.6 Approach Overview

A complete illustration of our approaches is shown in Figure 1.1.

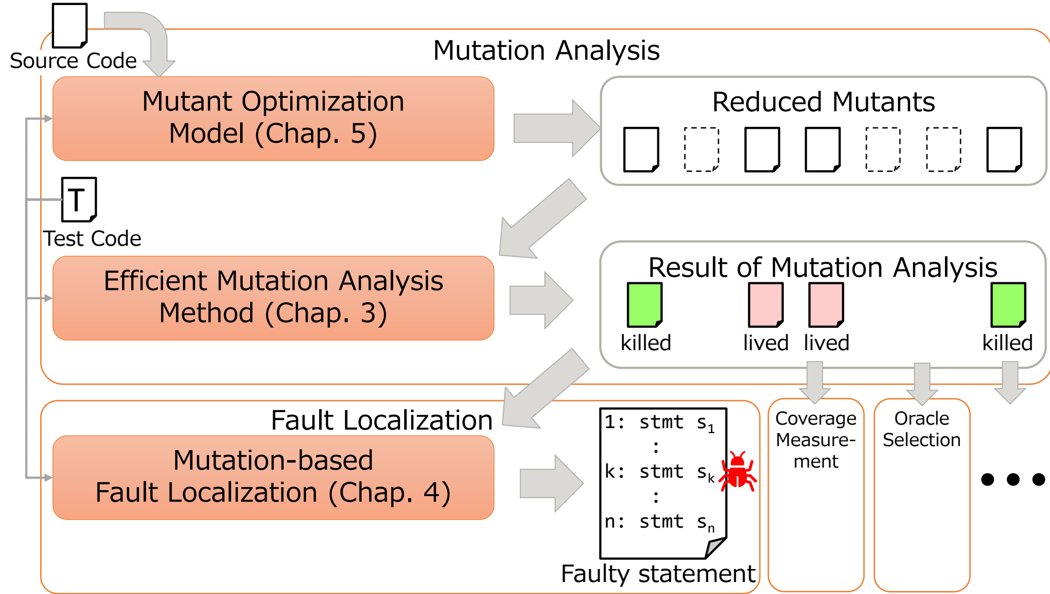

Figure 1.1: Overview of our approaches

First, mutant optimization reduces the number of mutants to be run, with minimal impact on the analysis results. Next, an efficient method of executing mutation analysis is used to rapidly deliver the results. Finally, the results of the mutation analysis can be applied to improve the performance of various test and debug technologies. We present fault localization as a promising case in point.

1.7 Contributions

The main contributions of this thesis are as follows:

- A method to reduce compilation cost by integration of bitcode mutation and metamutation.
- A method to reduce testing time by invoking a process once and splitting execution stream for higher order mutation.
- A method to reduce the number of mutants by an online adaptation technique that omits infeasible mutants.
- An empirical comparison between MuVM and existing techniques.
- A method for determining mutant redundancy by the type of errors caused by mutation. We then introduce a model that reduces mutants with overlapping types of errors by treating them as redundant.
- A method to quantify the impact of reduced mutants and score the remaining mutants. As a result, we are able to reduce the discrepancy between the mutation score after mutant reduction and that of the original mutant set.

- We propose and implement a mutation-based fault localization using statement deletion mutation, called SDL-MBFL.
- We compare fault localization techniques, including SDL-MBFL, using actual product software and multiple faults in an enterprise.
- In the above comparison, SDL-MBFL achieve the same number of fault detections and reduce the execution time by 20.3% compared to conventional MBFL.

1.8 Organization

The remainder of this thesis is organized as follows.

Chapter 2 We introduce background on software testing and debugging with mutation analysis, including coverage measurement and fault localization.

Chapter 3 We present the results of a systematic literature survey on coverage measurement in industrial software development in order to understand what the quality of coverage measurement essentially means in industrial software development.

Chapter 4 We present a mutation analysis tool called MuVM, which improves execution time significantly using four techniques: metamutation, mutation on a virtual machine, higher order split-stream execution, and online adaptation. Our experiments indicate that our tool is significantly superior to an existing tool.

Chapter 5 We show a performance improvement of mutation-based fault localization using a statement deletion operator implemented on MuVM, and we evaluate it on industrial software used in actual products and nine defects that actually occurred.

Chapter 6 We propose a technique for highly reliable mutation score measurement to prevent excessive pruning of mutants by selecting mutants covering kinds of errors caused during mutation analysis and by scoring selected mutants. As a result of evaluation using open source software (OSS), we achieve an approximate 40% reduction in execution time and show that the proposed technique enables a smaller difference in mutation score than the existing technique.

Chapter 7 We survey work related to our methods in the field of efficient mutation analysis, mutant optimization, mutation-based fault localization, and application of mutation analysis.

Chapter 8 Finally, we conclude this thesis and discuss directions for future work.

Chapter 2

Background on Mutation Analysis

This chapter reviews the fundamental concepts of mutation analysis.

2.1 Terminology of Software Problems

We clarify the definitions for terms related software defects.

According to Zeller [207], the word "bug" suggests something humans can touch and remove. As such, it is a term that lacks precision. Applied to programs, a bug can mean:

- An incorrect fragment of program code
- An incorrect program state
- An incorrect program execution

In order to prevent confusion, we avoid using the term "bug" in this paper and define the terms related to "bug" as follows

Definition 2.1 (Fault)**.** An incorrect step, process, or data definition in a program code (a bug in the code).

Definition 2.2 (Infection)**.** An incorrect program state (a bug in the state).

Definition 2.3 (Failure)**.** An observable incorrect program behavior (a bug in the behavior).

Figure 2.1 illustrates the relationship between these words, that is interpreted as *"The fault caused an infection, which caused a failure – and when we saw the failure we tracked the infection, and finally found and fixed the fault."*

The definition of "fault" above conforms to the IEEE Standard 610.12 [45] definition, while Zeller's book defines the term "defect" in the same sense as the term "fault" here.

The term "error" is similarly misleading, and could be used to refer to a fault, failure, or mistake. According to the IEEE standard 610.12, one of the semantic assignments is that the definition of "error" is "the difference between a computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition", and we follow the manner. The difference between "error" and "failure" is that "failure" focuses on the abnormal behavior of the program itself, while "error" focuses on the gap between the abnormal behavior of the program and the expected behavior.

Figure 2.1 illustrates the *RIP (Reachability, Infection, Propagation) model* which states that three conditions must be present for a failure to be observed.

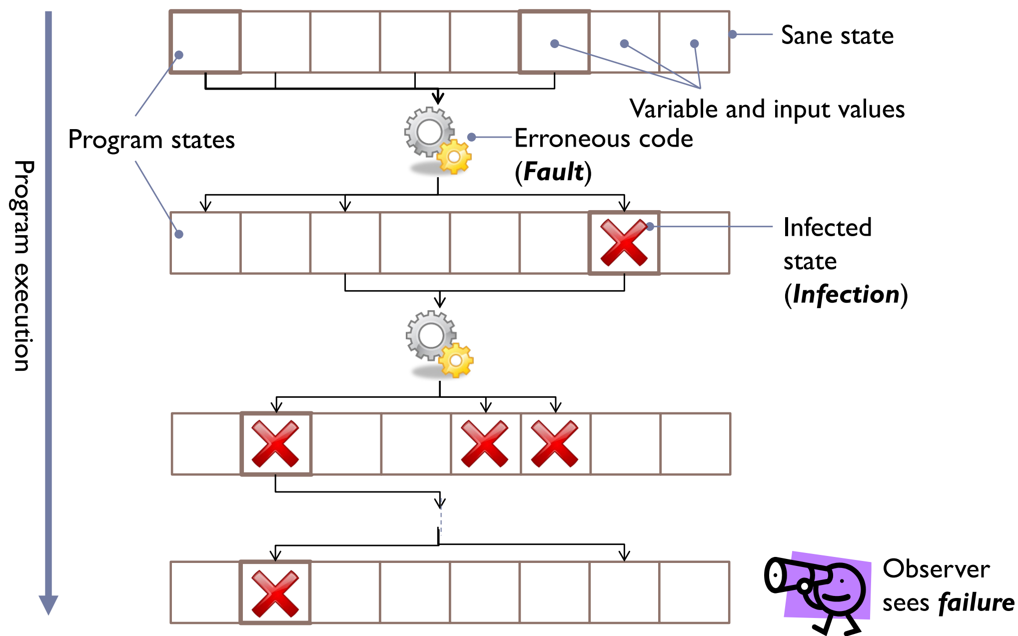

Figure 2.1: Terminology in Program failure

Reachability The location or locations in the program that contain the fault must be reached.

Infection After executing the location, the state of the program must be incorrect.

Propagation The infected state must propagate to cause some output of the program to be incorrect.

The RIP model implies that execution of a good test reaches a fault location, changes the state of the program into incorrect one, and propagate the infection to the exit of the program. The concept of the RIP model is important for the idea of mutation analysis, that will be explained in section 2.3.

2.2 Software Testing

In 1979, G. Myers defined testing as "Testing is the process of executing a program or system with the intent of finding errors". That means a good test should have a high probability of finding an error. In fact, practitioners might want their tests to find the most number of errors with a minimum amount of effort.

In figure 2.2, we illustrate our model of software testing. We define the salient terms referenced in the figure as follows.

Definition 2.4 (Test data)**.** Test data is input for the software under test.

Definition 2.5 (Test procedure)**.** A test procedure is a way of initialization, inputting test data, and checking the result, usually described as a document or code.

Definition 2.6 (Test oracle)**.** A test oracle is a mechanism of checking the result.

Definition 2.7 (Test case)**.** A test case is the minimum unit of test execution, which contains test data, a test procedure, and an expected result.

Definition 2.8 (Test suite)**.** A test suite is a collection of test cases that are usually grouped together for a specific purpose.

To test the software, each test case is executed on it. This entails inputting the test data into the software under test, according to the test procedure, executing the software, obtaining the output, and determining the pass or fail status of the test by comparing the output value with the expected value.

Although our software testing model can potentially be applied to both manual and automated testing, and to both unit and integrated testing, it is basically designed for automated unit testing because this thesis aims to improve the process of automated unit testing. For unit test code, test cases correspond to test methods, test oracles correspond to assertions, and test suites correspond to test classes or files of test code, etc.

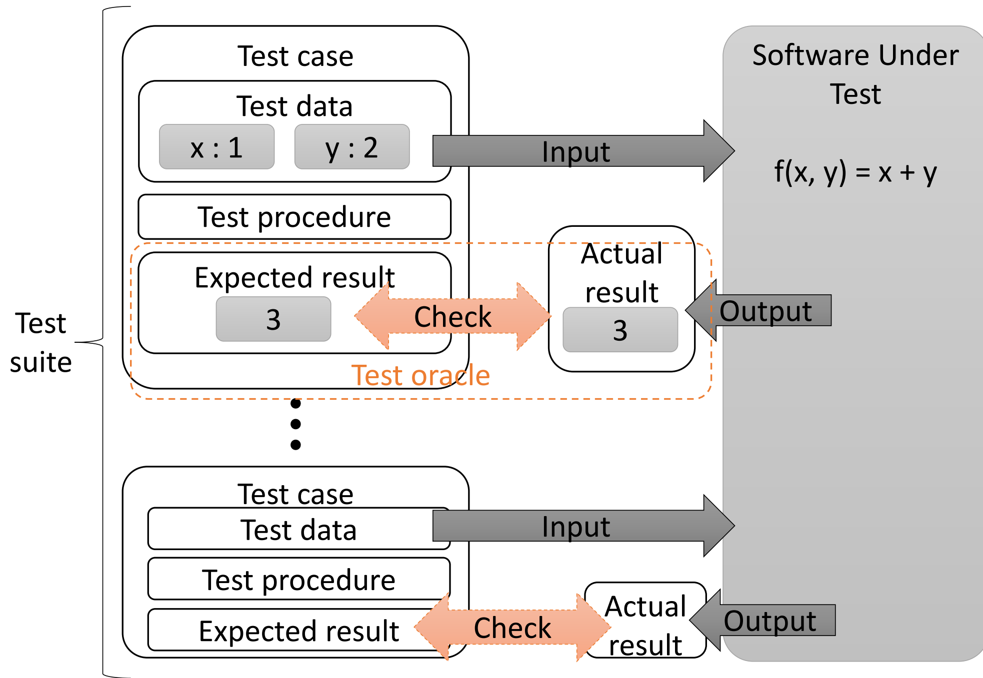

Figure 2.2: Test suites, test cases, test data and test oracle

To control the quality of tests it is essential to measure the fault detection capability of test suites. A formal coverage criterion serves this important purpose. For example, with a formal coverage criterion, practitioners can decide test inputs, and devise stopping rules for the testing process. Metrics based on code structure, such as statement coverage and branch coverage, are the most widely used code coverage criteria.

Ammann and Offutt [9] define coverage criteria in terms of test requirements. The basic idea is that the set of test cases are required to have a number of properties, each of which is fulfilled by specific individual test cases.

Definition 2.9 (Test Requirement)**.** A test requirement is a specific element of a software artifact (e.g. source code) that a test case must satisfy or cover.

Test requirements can be described with a variety of software artifacts, including the source code, design components, modeling elements, and the input space. A coverage criterion is simply a recipe for generating test requirements in a systematic way:

Definition 2.10 (Coverage Criterion)**.** A coverage criterion is a rule or collection of rules that impose test requirements on a test suite.

Using the above definitions we can define the coverage of a test suite as follows:

Definition 2.11 (Coverage)**.** Given a set of test requirements TR for a coverage criterion C, a test suite T satisfies C if and only if for every test requirement tr in TR, there exists at least one test case t in T such that t satisfies tr.

Note that it is acceptable to satisfy a given test requirement with more than one test case.

If the preparation cost of test data is too high, it may be hard to satisfy a coverage criterion completely. We use the notion of coverage level for quantifying the degree to which a test suite satisfies a coverage criterion.

Definition 2.12 (Coverage Level)**.** Given a set of test requirement TR and a test suite T, the coverage level is simply the ratio of the number of test requirements satisfied by T to the size of TR.

Coverage criteria are traditionally used in designing test cases. Test suites are designed to satisfy a given coverage criterion by iteratively measuring the coverage level and finding and preparing test cases that contribute to improving the coverage level.

2.3 Mutation Analysis for Assessing Test Quality

Structural code coverage is widely employed for evaluating the efficacy of a test suite, by measuring the fraction of program elements, such as statements and branches, that are covered by the execution of the test cases. However, code coverage does not evaluate whether the tests actually catch faults through test oracles [186], i.e., in terms of the RIP model, it only measures the reachability of the test suite. Mutation analysis [55, 88] is an alternative method for evaluating the fault-revealing capability of a test suite. It seeds faults artificially and calculates the percentage of the seeded faults the tests can detect. In other words, mutation analysis can examine the ability of a test to ascertain not only reachability but also infection and propagation.

Artificially inserted faults are defined as rules, called mutation operators, for how to alter a program, specific to each programming language. For example, a program element of C “`x + 2`” generates 4 altered program elements “`x - 2`”, “`x * 2`”, “`x / 2`”, and “`x % 2`” by a mutation operator called *arithmetic operator replacement* (OAAN).

A mutation operator and a mutant can be defined as follows.

Definition 2.13 (Mutation Operator)**.** A rule that specifies syntactic variations of a program element.

Definition 2.14 (Mutant)**.** The program resulting after one application of (basically) one mutation operator.

The earliest experiments on mutation analysis were done in the context of Fortran programs. The first formal definition of mutation operators for Fortran was given by Offutt and King [112, 155]. This definition has become the cornerstone of the mutation operators adopted by many tools. Today there are mutation operators for a variety of languages. [3] defined a set of mutation operators for

C such as statement mutations, operator mutations, variable mutations and constant mutations. In Java, PIT is the most widely used mutation analysis tool. We employ it in Chapter 6 of this thesis. PIT's mutation operators bear many similarities to traditional mutation operators, but differ in their design. Table 2.1 shows a comparison of mutation operators for Fortran, C, Java.

Table 2.1: A comparison of mutation operators for Fortran [112], C [3], and Java (PIT) [44]

Fortran 77 Operator	Description	C operator	Java (PIT) operator
AAR	Array reference for array reference replacement	VLSR, VGSR	—
ABS	Absolute value insertion	VDTR	ABS
ACR	Array reference for constant replacement	VLSR, VGSR	—
AOR	Arithmetic operator replacement	OAAN	AOR, Math
ASR	Array reference for scalar variable replacement	VLSR, VGSR	—
CAR	Constant for array reference replacement	CGSR, CLSR	—
CNR	Comparable array name replacement	VLSR, VGSR	—
CRP	Constant replacement	CRCR	Constant Replacement
CSR	Constant for scalar replacement	CGSR, CLSR	—
DER	DO statement END replacement	OTT	—
DSA	DATA statement alterations	—	—
GLR	GOTO label replacement	SGLR	—
LCR	Logical connector replacement	OBBN	OBBN
ROR	Relational operator replacement	ORRN	ROR, Conditionals Boundary, Negate Conditionals
RSR	Return statement replacement	SRSR	—
SAN	Statement Analysis	STRP	—
SAR	Scalar variable for array reference replacement	VLSR, VGSR	—
SCR	Scalar for constant replacement	VLSR, VGSR	—
SDL	Statment deletion	SSDL	Void Method Call, Remove Conditionals, Remove Increments
SRC	Source constant replacement	CRCR	Constant Replacement
SVR	Scalar variable replacement	VLSR, VGSR	—
UOI	Unary operator insertion	OLNG, VTWD	UOI

Mutation analysis executes the test suite for each generated mutant. If one or more of the test cases fails, the mutant is said to be "killed by the test cases or test suite". The more mutants, out of the generated population of mutants, a test suite kills the better it is judged to be.

However, there are some mutants that cannot be killed by any possible test case. These are mutants whose behavior is the same as the original program. Such a mutant is termed an equivalent mutant. For example, if the original program is "`x = 2 + 2`", then the mutant "`x = 2 * 2`" is an equivalent mutant.

Recalling the definition of coverage level, we can determine a coverage level for mutation analysis by assuming that the set of test requirements is to kill all mutants. This coverage level is called the mutation score and is defined as follows.

Definition 2.15 (Mutation Score)**.**

$$Mutation\,Score = \frac{\#\,of\,killed\,mutants}{(\#\,of\,all\,mutants) - (\#\,of\,equivalent\,mutants)} \tag{2.1}$$

However, since the detection of equivalent mutants is an undecidable problem, proved by Budd and Angluin [31], and beyond the scope of this thesis, for the rest of the thesis we calculate the mutation score without considering the number of equivalent mutants (i.e., as $\frac{\#ofkilledmutants}{\#ofallmutants}$).

2.4 Fundamental Hypotheses

The concept of mutation analysis is based on two hypotheses.

- Competent Programmer Hypothesis (CPH)
- Coupling Effect

CPH is a hypothesis proposed by DeLilo et al. [55] in 1978, which states that a programmer is competent enough to write almost perfect source code. In other words, a truly correct program is only marginally different from one written by a programmer. Therefore, introducing a slight change in the program would simulate inserting a realistic fault. As evidence to support CPH, Gopinath et al. [87] mined faults and their corrections in the repository and investigated the characteristics of the faults; they found many faults with only a single token, while typical faults contained three or four tokens.

The notion of Coupling Effect was also proposed by DeMillo et al. [55] in 1978 and states that: "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". This assumption was extended by Offutt [154, 156] for mutation analysis as follows: "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants". This means that if a test suite can be designed to detect simple mutants, it will also have the capability to detect a significant number of complex defects.

2.5 Process of Mutation Analysis

When a mutation analysis is used to evaluate a test suite, developers typically use the results of that mutation analysis to improve the test suite. Figure 2.3 shows a process, a modification of the one originally described by Usaola and Mateo [197], where developers use mutation analysis to incrementally improve the test suite.

The inputs are a test suite, a program under test, and a mutation score threshold, which is the process termination criteria. The process is broadly divided into three phases: test execution, test evaluation, and test improvement, respectively.

In the first phase, test execution, the developer runs the test suite against the program under test to make sure that all test cases pass. If the test case fails, the program under test is modified so that it passes the test.

In the next phase, test evaluation, mutants are generated and executed on the program under test. The mutants to run are either for newly added or modified parts of the program, or for mutants that have not been killed in previous mutant runs.

If the mutation score obtained from the results exceeds a given threshold, this process is terminated; otherwise, in the third phase – test improvement – the developer investigates the survived mutants from the results of the mutation analysis and adds test cases that can kill them, i.e., improve the mutation score. The developers test the program again with the improved test suite to catch the

new faults. They repeat the cycle of test execution, test evaluation, and test improvement, and eventually obtain a sufficiently high quality test suite that the mutation score exceeds the threshold.

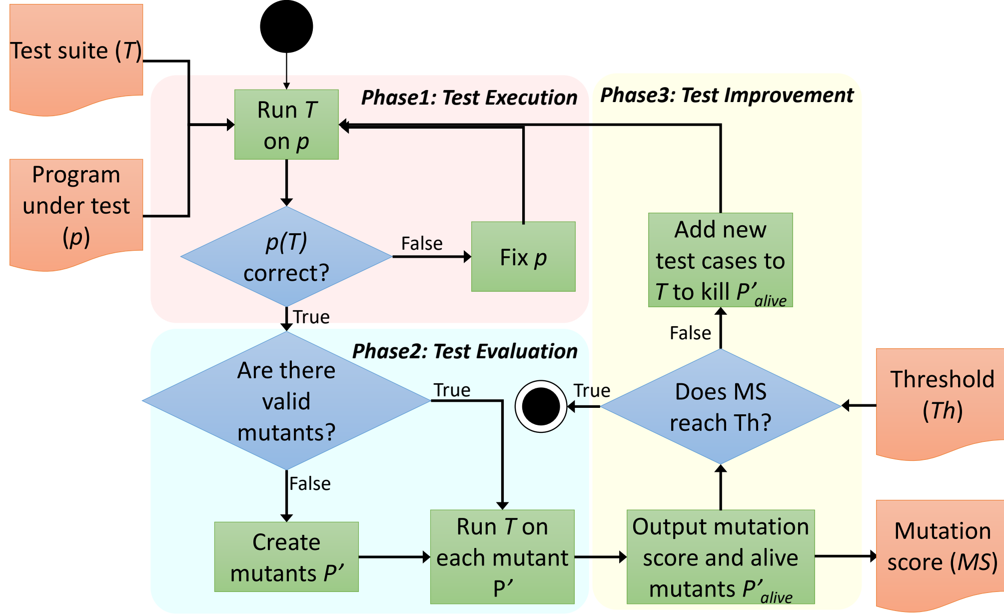

Figure 2.3: A process of mutation testing for incremental improvement of a test suite

2.6 Computational Cost of Mutation Analysis

The computational cost of mutation analysis is very high due to the large number of mutants that need to be analyzed. Mutants are generated exhaustively, corresponding to each program element to which mutation operators can apply. For example, as shown in Section 2.3, "`x + 2`" can generate 4 mutants "`x - 2`", "`x * 2`", "`x / 2`" and "`x % 2`" by just one OAAN mutation operator. The total number of generated mutants increases with the number of applied mutation operators and the size of the program. The number of mutants directly affects the running time of mutation analysis. Fig. 2.4 illustrates the cost of traditional mutation analysis. The runtime T_{total} of mutation analysis can be expressed as:

$$T_{total} = \sum_{m \in M} t_{seed,m} + \sum_{m \in M} t_{cmpl,m} + \sum_{m \in M} \sum_{tc \in TC} t_{test,m,tc}$$

where M is the set of mutants, TC is the set of test cases, and $t_{seed,m}$, $t_{cmpl,m}$ and $t_{test,m,tc}$ are the times for seeding (mutating), compiling, and testing respectively, a combination of a mutant m and a test case tc . Generally speaking, the compilation time and the testing time are the dominant components in the total time of traditional mutation analysis.

According to [99], cost reduction techniques are roughly categorized into two approaches. One is a mutant reduction technique which reduces the size of M, and the other is an execution cost reduction technique which reduces the seeding time T_{seed}, the compiling time T_{cmpl}, and the testing time T_{test}.

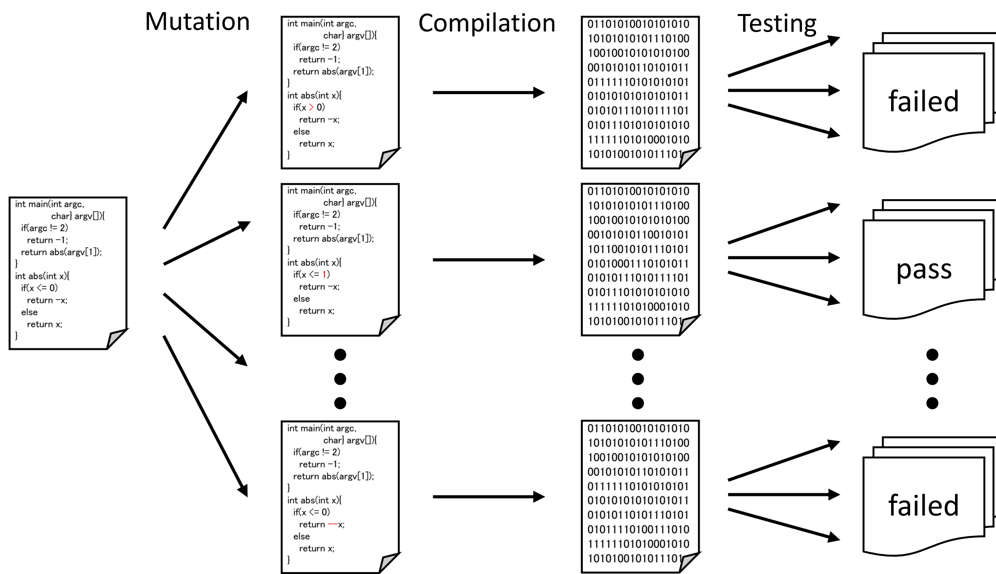

Figure 2.4: Computational cost of mutation analysis

Chapter 3

A Systematic Literature Review of Code Coverage Measurement in Industrial Testing

3.1 Overview

Code coverage is an indicator to check the sufficiency of the tests for the target source code based on the defined criteria. It has a long history, and since the first proposal by Miller and Maloney[138] in 1963 many coverage measurement techniques including mutation analysis have been proposed, and many tools exist. In addition, coverage measurement techniques are widely used in industry. Not only the coverage itself, but also the quality of such coverage measurement techniques and tools is an important factor that affects the productivity of developers. Against this background, many techniques for the quality of coverage measurement techniques and tools have been proposed. However, to the best of our knowledge, there is no literature that investigates what kind of quality is required for coverage measurement in industrial software development. In this chapter, we conducted a systematic literature review on coverage measurement in industrial software development in order to understand what the quality of coverage measurement essentially means in industrial software development.

3.2 Research Method

In this section, we describe the method of systematic literature review (SLR). Figure 3.1 shows the overview of our systematic literature review.

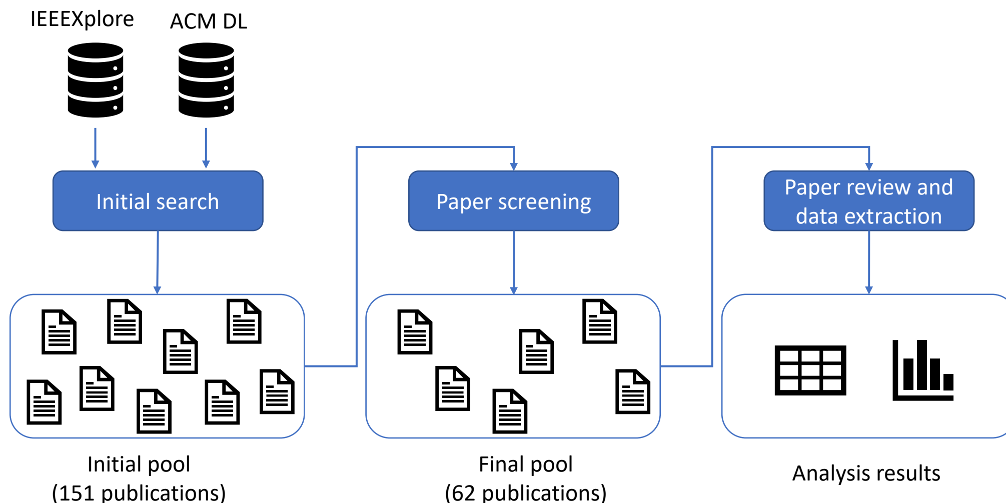

Figure 3.1: Research process used to conduct this study

3.2.1 Goal and Research Questions

First, we defined the purpose of the survey and the research questions.

The major purpose of this survey is to clarify what coverage and mutation analysis essentially mean in industrial software development. Specifically, the following is described.

- What is the impact of coverage and mutation analysis on the quality of industrial software?
- How does the quality of the coverage and mutation analysis tools themselves affect the quality of industrial software?
- What kind of quality and functionality is required of the coverage and mutation analysis tools themselves?

The following research questions were formulated for these purposes.

- RQ1: Which programming languages of software under test (SUT) are popular for coverage measurement?
- RQ2: What types of coverage criteria are used?
- RQ3: For what purpose is coverage used?
- RQ4: What effects have resulted from the use of coverage?
- RQ5: What quality characteristics are required in coverage measurement tools?

The purpose of RQ1 and RQ2 is to understand the language and the type of coverage criteria used in the industry as background. The purpose of RQ3 is to identify what coverage was used for in the testing activities (e.g. test generation, test assessment). RQ4 clarifies what effects have occurred as a result of testing activities using coverage (e.g., fault detection, test suites improvement). RQ5 examines what quality characteristics are required for the coverage measure itself and what impact it has.

3.2.2 Research Process

In order to clarify the research questions defined above, we collected literature and conducted a survey of it. The following search engines were used to collect the literature.

1. IEEEXplore
2. ACM Digital Library

We searched the literature on coverage in industrial software testing using the queries shown in Table 3.1 in each search engine and obtained 151 publications.

From the 151 publications obtained in the initial search, we selected documents that were appropriate for our purpose. For the selection, we manually checked the contents and selected the references according to the following criteria.

- Dealing with software developed as an activity of a company
- Dealing with coverage measurement through testing of software

Table 3.1: Search keywords and number of publications

Search engines	Query	# of publications
IEEE Xplore	(”Document Title”:”code coverage” OR ”Abstract”:”code coverage” OR ”Author Keywords”:”code coverage”) AND ((”Document Title”:industry OR industrial) OR (”Abstract”:industry OR industrial) OR (”Author Keywords”:industry OR industrial))	91
ACM Digital Library	(Title:(”code coverage”) OR Abstract:(”code coverage”) OR Keyword:(”code coverage”)) AND (Title:(industry OR industrial) OR Abstract:(industry OR industrial) OR Keyword:(industry OR industrial))	44

For example, papers in which authors affiliated with a company measure coverage of in-house software are included, but papers dealing with general OSS and experimental programs as coverage measurement targets are excluded. Note that even OSS whose main activity is in companies is included.

After this selection process, 62 of the 151 publications met the criteria. Table 3.2 and 3.3 contain the details of all 62 publications. Some blank cells in the table, e.g. language or application area, mean that the information is not known from the publication. Number of the publications per year is shown in Figure 3.2. Some of the older publications are from the 1990s, but the number was not large until 2010, and has been increasing since the beginning of the 2010s. One possible reason for the increase in recent years is the proliferation of coverage measurement tools in the industry.

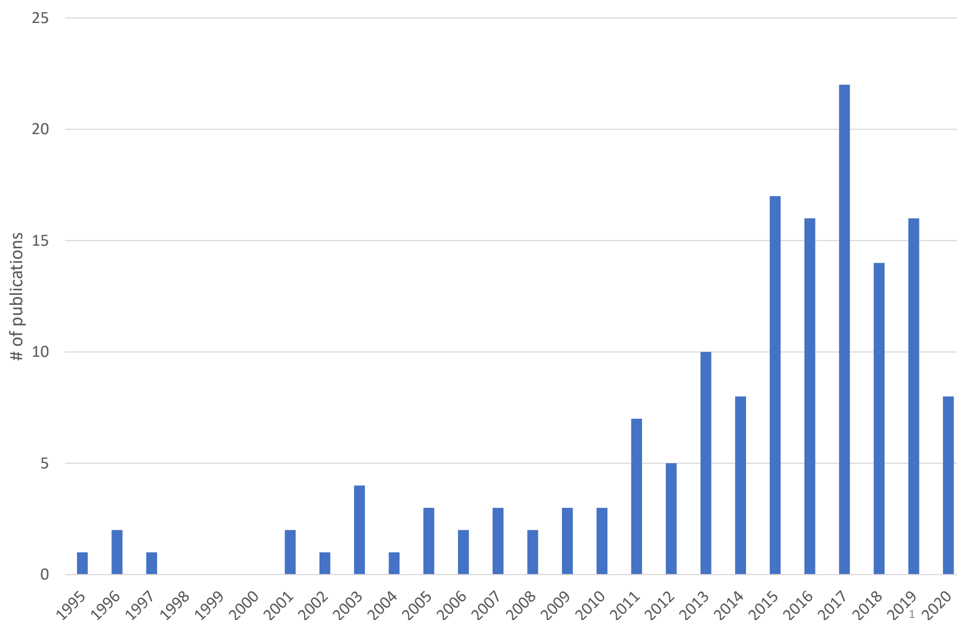

Figure 3.2: Annual trend of publications including industrial coverage measurement

Table 3.2: All relevant publications in this study and their coverage data

Author(s)	Publication Year	Language	Application area of SUT	Scale of SUT	Statement/ line/ block coverage	Branch/ decision coverage	Method coverage	Mutation score	MC/DC	Specification/ requirement coverage	Other coverage type	Purpose of coverage	Effects of coverage
H. Bergström and E. P. Enoiu[23]	2017	PLC	Train control management system		0	0	0	0	0	0	timed base-choice criteria	Test generation	fault detection
B. Chen[38]	2019	Java	Cloud service, Video streaming, Distributed file system, API library		1	1	1	0	0	0		Test assessment	improve test suites
E. Enoiu et al.[68]	2017	PLC	Industrial Control Software		0	1	0	0	0	0		Test generation	fault detection
P. Charbachi et al.[36]	2017	PLC	Train control management system		0	1	0	1	0	0		Test assessment	evaluation
Y. Adler et al.[2]	2011	Java	IBM products		0	0	1	0	0	0		Test assessment	improve test suites
R. Carlson et al.[34]	2011	C++	ERP package		0	0	1	0	0	0		Test prioritization	fault detection
W. Wang et al.[199]	2018	Java	Android app		0	0	1	0	0	0		Test assessment	evaluation
T. Bach et al.[17]	2017	C, C++	SAP HANA		1	0	0	0	0	0		Test assessment	fault detection
J. McDonald et al.[137]	2001	C, C++	Firmware		1	1	0	0	0	0		Test assessment	fault detection
B. K. Papis et al.[167]	2020	C#	Web services supporting gas market and underground gas storage facility operations		1	0	0	0	0	0		Test assessment	evaluation
W. E. Wong and J. Li[203]	2005	Java	Avaya product		1	1	0	0	0	0	all-primary-nodes, all-secondary-nodes, all-primary-edges, all-secondary-edges	Test assessment	improve test suites
J. Slonim et al.[184]	1996	C			1	1	0	0	0	0	all-use	Test assessment	improve test suites
T. Bach et al.[16]	2017	C, C++	SAP HANA		1	0	0	0	0	0		Test optimization	test redundancy detection
Mei-Hwa Chen et al.[41]	1996	C	Automatic flight control project		1	0	0	0	0	0		Test assessment	reliability estimation
T. Gergely et al.[79]	2010		Financial system	631,043 LOC	0	0	1	0	0	0	coverage measurements based on procedure call and control transfer	Test optimization	test redundancy detection
J. Blom et al.[26]	2016	C, Erlang	Mobile Network System	130,000 LOC (Erlang) + 5,500 LOC (C)	1	0	0	0	0	0	model-based coverage	Test assessment	fault detection
C. Klammer et al.[117]	2018	Java	UI framework	280kLOC	0	0	1	0	0	0		Test assessment	monitoring test quality
H. Hemmati et al.[91]	2018	C	Unmanned Aerial Vehicle	4,843 functions and 66,242 C/Ds	0	0	0	0	1	0		Test assessment	fault detection
X. Qu et al.[175]	2012	C, C++	real-time embedded system at ABB	1.18MLOC	0	0	1	0	0	0	changed function coverage	Test assessment	evaluation
G. Petrovic and M. Ivankovic[170]	2018	C++, Java, Python, Go, JavaScript, TypeScript, Common Lisp		72,425 diffs	1	0	0	1	0	0		Test assessment	improve test suites
L. Hao et al.[89]	2019	PLC	Bombardier Transportation	5706 LOC	1	1	0	0	0	0		Test generation	evaluation
D. Amalfitano et al.[7]	2015	Java	a garage management system and an order management system		1	1	0	0	0	0		Test assessment	test adequacy
I. Nica et al.[148]	2017	C++	Computer Vision Library	100,000 LOC	1	1	1	0	0	0		Test assessment	evaluation
J. C. Cunha et al.[46]	2012	C	Space		1	1	0	0	1	0		Test assessment	test adequacy check
Mei-Hwa Chen et al.[42]	1997	C	Automatic flight control project		1	0	0	0	0	0		Test assessment	reliability estimation
S. Huang et al.[96]	2019		Android App		0	0	1	0	0	0		Test assessment	evaluation
B. Chen et al.[39]	2018	Java	Cloud service, Video streaming, Distributed file system, API library		1	1	0	1	0	0		Test assessment	improve test suites
A. Eriksson and B. Lindström[69]	2016	C++	avionics	637 classes	0	1	0	0	0	0	All Navigation (ANAV), All Iteration (ITER)	Test assessment	reliability estimation
A. S. Dookhun and L. Nagowah[66]	2019	Java	the String Calculator and the Bowling Score Keeper		0	1	0	0	0	0		Test assessment	evaluation
D. Di Nardo et al.[62]	2015	Java	data acquisition system	32170 bytecode instructions	1	0	0	0	0	0		Test generation	evaluation

Table 3.3: All relevant publications in this study and their coverage data (cont.)

Author(s)	Publication Year	Language	Application area of SUT	Scale of SUT	Statement/ line/ block coverage	Branch/ decision coverage	Method coverage	Mutation score	MC/DC	Specification/ requirement coverage	Other coverage type	Purpose of coverage	Effects of coverage
P. Luchscheider and S. Siegl[129]	2013	MATLAB/ Simulink	Automotive thermal control unit		0	1	0	0	0	0		Test assessment	evaluation
M. -. Chen et al.[40]	2001		automatic flight control and space	7000 LOC and 10000 LOC	1	0	0	0	0	0		Test assessment	reliability estimation
J. Lawrence et al.[120]	2005	C#		10 methods	1	0	0	0	0	0		Test assessment	test adequacy check
S. Berner et al.[24]	2007	Java	Bank's backend system	30 50 KLOC	1	1	1	0	0	0		Test assessment	fault detection
S. M. B. Bhargavi et al.[25]	2016		retail		0	0	0	0	0	0	combinatorial coverage	Test assessment	test adequacy check
H. Nakagawa et al.[145]	2017	Java	Web application for inventory management	263 specification items and 2,181 test case descriptions	0	0	0	0	0	1		Test assessment	test adequacy check
R. Gerlich and C. R. Prause[80]	2018	C	spacecraft		1	1	0	0	0	0		Test generation	termination of test generation
J. Czerwonka[47]	2013		Windows utilities		1	1	0	0	0	0		Test assessment	evaluation
Y. Sun et al.[188]	2017	C, Ada	avionics		1	0	0	0	1	0		Test assessment	evaluation
G. Buchgeher et al.[30]	2013	C, C++, C#, VB	electrical hardware diagnostics	2.5MLOC	1	1	0	0	0	0		Test assessment	test redundancy detection
S. Tokumoto and K. Takayama[0]	2019	Java	factory automation	37KLOC	0	0	0	1	0	0		Test assessment	test adequacy check
G. Fraser and A. Arcuri[75]	2013	Java		809 LOC	0	1	0	0	0	0		Test generation	test redundancy detection
A. Arcuri[12]	2017	Java	Web services	3584 LOC	1	0	0	0	0	0		Test generation	test redundancy detection
H. Nakagawa et al.[146]	2017	Java	Web application for inventory management	263 specification items and 2,181 test case descriptions	0	0	0	0	0	1		Test assessment	test adequacy check
G. Fraser and A. Arcuri[74]	2011	Java	real-time embedded system		0	1	0	0	0	0		Test assessment	test redundancy detection
S. K. Khalsa and Y. Labiche[109]	2016		a middleware functionality in the telecom domain (Ericsson)	694 LOC	1	1	0	1	0	0	Constrained Base Choice (CBC), Extended Constrained Base Choice (ECBC)	Test assessment	fault detection
K. Ramasamy and S. Arul Mary[176]	2008	VB, PHP		5000 LOC (VB), 6000 LOC (PHP)	0	0	0	0	0	1		Test prioritization	fault detection
H. Hemmati et al.[92]	2015		Mozilla Firefox		0	0	0	0	0	0	topic coverage	Test prioritization	fault detection
S. Siegl et al.[183]	2015		automotive		0	1	0	0	0	0	State coverage, Transition coverage	Test assessment	evaluation
M. Ivanković et al.[98]	2019	C++, Java, Python, Go, JavaScript, Dart, TypeScript	Services in Google	1 billion LOC	1	1	0	0	0	0		Test assessment	test adequacy check
Y. W. Kim[111]	2003	C++, Java		19,800 KLOC	1	1	0	0	0	0		Test assessment	test adequacy check
C. Magalhães et al.[133]	2017		Android app		0	0	1	0	0	0		Test assessment	evaluation
X. Zeng et al.[208]	2016	Java	WeChat Android app	610,629 LOC	1	0	0	0	0	0	Activity coverage	Test assessment	evaluation
A. Eriksson et al.[70]	2012	C++	aviation applications for Saab Gripen fighter	637 classes	0	0	0	0	0	0	Clause coverage (CC), Predicate coverage (PC), Correlated Active Clause Coverage (CACC)	Test assessment	improve test suites
R. Ramler et al.[177]	2017	C	mechatronic system	57,363 LOC	0	0	0	1	1	0		Test assessment	improve test suites
M. Kessis et al.[108]	2005	Java	J2EE server	200,000 LOC	1	0	1	0	0	0	condition coverage	Test assessment	test adequacy check
D. Di Nardo et al.[62]	2015	Java	data processing system	32,170 bytecode instruction	1	0	0	0	0	0		Test assessment	evaluation
T. Cai et al.[33]	2020		Mobile App		1	0	0	0	0	0	Activity coverage	Test assessment	evaluation
A. Windisch et al.[201]	2007	C	automotive	4414 LOC	1	1	0	0	0	0	condition coverage	Test generation	evaluation
R. Taylor and J. Derrick[189]	2015	Erlang	SAT¿IP component (satellite TV signal to IP)		1	0	0	0	1	0		Test assessment	fault detection
M. Gittens et al.[81]	2002	C++		201,629 LOC	1	0	0	0	0	0		Test assessment	evaluation
D. Di Nardo et al.[63]	2017	Java	data acquisition system	32,469 bytecode instructions	1	1	0	0	0	0		Test assessment	evaluation

3.3 Results

In this section, we show the results of the SLR.

3.3.1 RQ1: Which programming languages of SUT are popular for coverage measurement?

The number of selected publications per language is shown in Figure 3.3. The largest number is for Java, but the number of publications for C and C++ combined exceeds that for Java. These languages have been widely used in industry for a long time, and there are many coverage measurement tools available. Other languages include Python, Go, JavaScript, MATLAB/Simulink, Ada, and VB.

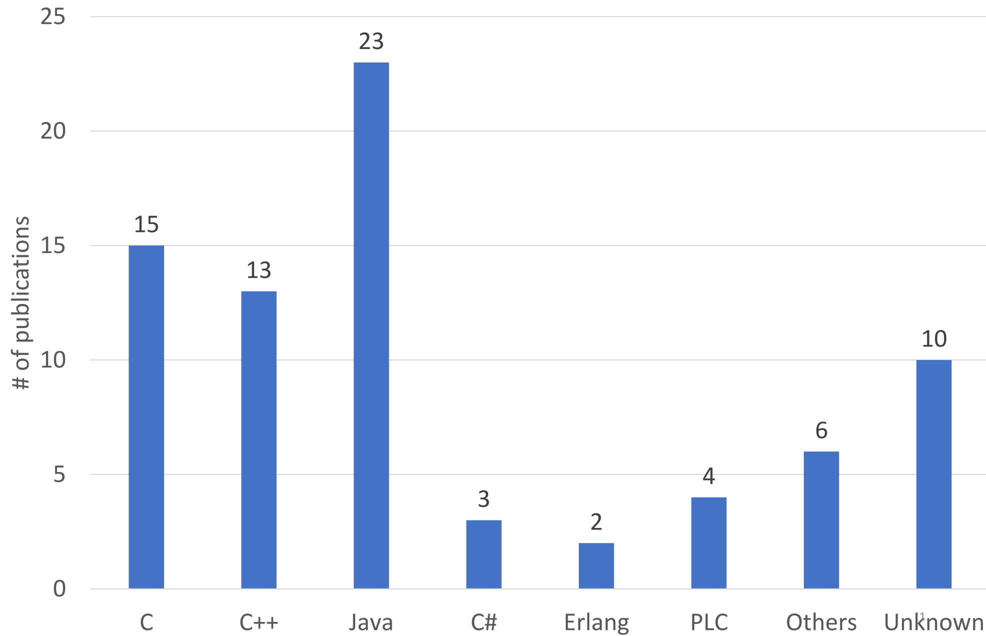

Figure 3.3: Number of publications by programming languages

3.3.2 RQ2: What types of coverage criteria are used?

The number of each coverage criteria for the selected publications is shown in Figure 3.4. Statement/line/block coverage is the most common, followed by branch/decision Coverage. Mutation score is used in only 6 publications. This result is considered to be due to the fact that most of the coverage criteria supported by the coverage measurement tools are statement/line/block coverage and branch/decision coverage, which are also the coverage criteria used in companies, while mutation score and MC/DC, which are stricter coverage criteria than other listed ones, are used only for software that requires high reliability. The other coverage criteria include model-based coverage, combinatorial coverage, condition coverage, and newly proposed coverage criteria.

3.3.3 RQ3: For what purpose is coverage used?

We classified the purpose of using the coverage in each publication into test assessment, test generation, test prioritization, and test optimization. The result is shown in Figure 3.5. Nearly 80% of the publications are for test assessment,

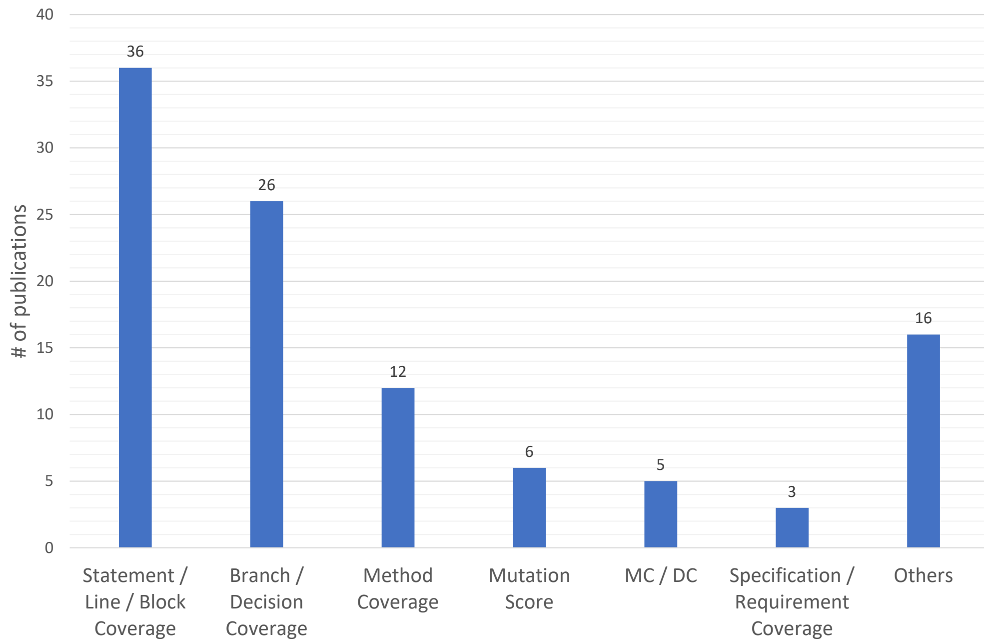

Figure 3.4: Number of publications by coverage criteria

which is considered to be the standard usage in coverage measurement tools. On the other hand, test generation, test prioritization, and test optimization account for about 20% of the publications, and their purpose is to process the test suite using the measured coverage information, which often requires a separate tool for that purpose. Most of the publications on test generation, test prioritization, and test optimization are proposals for such test suite processing methods, and we think that these methods are not yet in general use in industry.

3.3.4 RQ4: What effects have resulted from the use of coverage?

In the selected publications, the effects of coverage measurement were classified into eight categories: method evaluation, fault detection, test adequacy check, improving test suites, test redundancy detection, reliability estimation, monitoring test quality and termination of test generation. The largest category is method evaluation, which accounted for 32% of the total, followed by fault detection, which accounted for about 20%. The effect of method evaluation is not the effect of engineering, but rather the effect of verifying the hypothesis about coverage in experiments and surveys.

Regarding the confirmation of test sufficiency, coverage is considered to play the role of a proxy metric for test sufficiency, which cannot be measured directly as a metric. The effectiveness of coverage on defect detection has been shown as a result of empirical studies on the relationship between coverage and defect detection, as well as the fact that more defects can be detected by searching with coverage as a fitness function in test generation.

3.3.5 RQ5: What quality characteristics are required in coverage measurement tools?

We categorized what quality characteristics are required by the coverage measurement tools in the coverage measurements made in the selected publications. We counted the quality characteristics mentioned in the motivation, evaluation

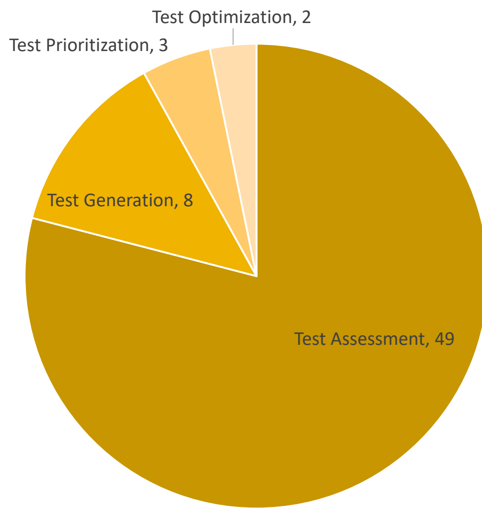

Figure 3.5: Number of publications by purpose

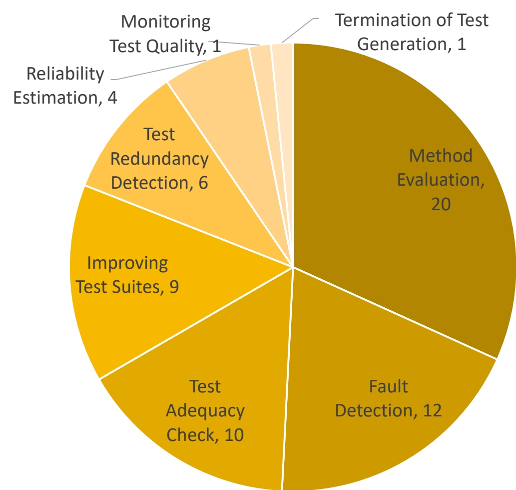

Figure 3.6: Number of publications by effect

and discussion within the publications.

The quality characteristics listed are speed, accuracy, usability, scalability, reliability, and memory consumption. Here are some examples for each quality characteristic. The example of speed is the challenge of the overhead involved in measuring coverage compared to normal test execution. The accuracy example is the challenge of the gap between the approximate coverage value and the true coverage value when calculating the approximate coverage value as a tradeoff for increasing other quality characteristics such as speed and usability. An example of usability is the challenge of assisting the user in checking the results of coverage measurements and facilitating a series of actions such as improving the test suite. An example of scalability is to complete the coverage measurement in a time that is acceptable to the developer, regardless of the size of the software to be measured. An example of reliability is to be able to measure coverage stably regardless of the type of software to be measured. An example of memory consumption is the issue of increasing memory consumption as more and more data is required for coverage calculation during coverage measurement.

The number of publications with and without mention of quality characteristics is shown in Figure 3.7 and the number of quality characteristics in publications with mention of quality characteristics is shown in Figure 3.8.

As a result, there is most often no mention of the quality characteristics of coverage measurement tools. When there is a mention of quality characteristics, most of the papers asked about usability and speed. For speed, the description is mainly about the overhead of instrumentation for coverage measurement.

Table 3.4 shows a list of literature with mention of quality characteristics.

Chen et al. [39, 38] improved performance by estimating coverage from program execution logs and eliminating the overhead of coverage collection. In its evaluation, they interviewed QA Engineers about the usefulness of the tool compared to existing tools.

Wong and Li (2005) [203] introduced a coverage measurement tool to an Avaya project at a low cost. It helps programmers and testers by providing quantitative visualization of the testing process. They also mentioned runtime overhead and memory consumption as challenges for the tool. In addition, the

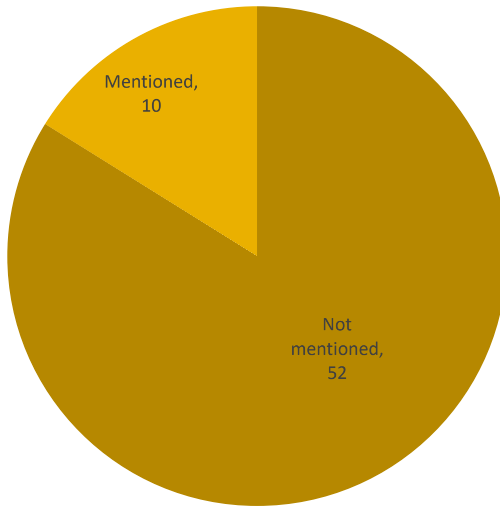

Figure 3.7: Number of publications by quality characteristics

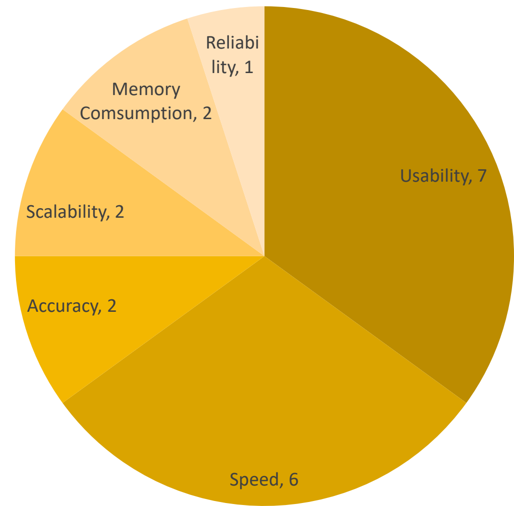

Figure 3.8: Number of quality characteristics in publications mentioned

trade-off between bytecode coverage and source code coverage is also addressed. In bytecode coverage, it is difficult to know what part of the source code is covered. However, it does not misrepresent covered instructions as not being covered. Source code coverage, on the other hand, allows us to know directly where the source code is covered. However, it is necessary to measure the coverage in bytecode first, and then map it to source code coverage. At that time, there is no one-to-one correspondence between bytecode and source code, so there may be a gap between the actual execution and the output source code coverage.

Slonim et al. (1996) [184] has applied a coverage tool called ATAC tool to industrial software. The requirements for industrial use are that the tool can be used for large-scale software, and that it must be reliable and stable. The lessons learned from the application are that the coverage tool should help to create additional test cases.

Gergely et al. (2010) [79] measured the completeness and redundancy of system-level tests using coverage measurement and change impact analysis, and used this information to redesign test cases and improve the efficiency of the testing process. The reduction of test cases is achieved by measuring the redundancy of the test cases by comparing the function call graphs between the test cases. The impact of the reduction on the coverage was shown by comparing it to the coverage obtained without the reduction, and the difference was small enough.

Ivanković et al. (2018) [98] investigated 5 years of coverage in Google's automated testing platform and analyzed 512 responses from a questionnaire survey of developers. They address the challenges of speed overhead for instrumentation, increased memory consumption, and flakiness.

Kim (2003) [111] investigated the relationship between coverage and metrics such as defect distribution in large scale projects. The challenges of coverage analysis in large-scale industrial software include the CPU-intensive nature of the analysis, the need for instrumentation, and the extra cost to testers. It was suggested that it is undesirable and cost-prohibitive to perform a detailed coverage analysis for all modules.

Ramler et al. (2017) [177] applied mutation testing in the C source code of 60KLOC, a safety-critical mechatronic embedded system. In their study, muta-

tion testing required more than 4,000 hours of execution on a PC cluster, and apart from the cost of setting up and maintaining an automated mutation testing process, it also required a lot of man-hours for developers to review the survived mutants and improve the tests.

Kessis et al. (2005) [108] analyzed the results of coverage measurements against 200KLOC of J2EE server middleware. They compared nine different coverage analysis tools in terms of supported coverage criteria, integration into project builds, reporting capabilities, and instrumentation approaches. They identified integration and instrumentation as the two major issues in coverage tools.

Taylor and Derrick (2015) [189] has developed a MC/DC measurement tool for Erlang. The tool generates HTML coverage reports and allows developers to interactively know the details of various levels of coverage.

Table 3.4: List of publications related to coverage measurement quality

Publication	speed	accuracy	usability	scalability	reliability	memory consumption
Chen et al. (2018) [39]	✓		✓			
Chen (2019) [38]	✓		✓			
Wong and Li (2005) [203]		✓	✓			✓
Slonim et al. (1996) [184]		✓	✓	✓	✓	
Gergely et al. (2010) [79]	✓					
Ivanković et al. (2018) [98]	✓					✓
Kim (2003) [111]	✓					
Ramler et al. (2017) [177]	✓		✓			
Kessis et al. (2005) [108]			✓	✓		
Taylor and Derrick (2015) [189]			✓			

3.4 Discussion

3.4.1 Context Type of Quality Characteristics

In the papers we reviewed, we found relevant descriptions of quality characteristics such as speed, accuracy, usability, scalability, reliability, and memory consumption. However, the context of the descriptions is different in each paper. There are three types of contexts for quality characteristic descriptions.

1. "Survey" type: quality characteristics that are found to be required as a result of the survey
2. "Research question" type: quality characteristics that are addressed and evaluated as research questions
3. "Future work" type: quality characteristics as future work that could not be tackled in the paper

"Survey" types include the speed challenges derived as a result of Kim[111]'s empirical study on coverage, and the speed and memory consumption challenges identified in Ivanković et al.[98]'s survey on coverage measurement in Google. The quality characteristics of "survey" type are statistically indicated by the importance of the issues related to them.

“Research question” types include methods to improve speed and memory consumption by Gergely et al.[79] and usability in tools by Taylor and Derrick[189]. The quality characteristics of “Research question” type are those that can be improved by the proposed method or trade-off effects occur in the research. This implies that the quality characteristics are important issues that should be solved to the extent possible with state-of-the-art techniques.

“Future work” types include the speed and memory consumption issues raised by Wong[203] in their introduction of a coverage measurement tool to a real system development project, and the issues raised by Ramler et al.[177] identified speed and usability as challenges when introducing their mutation testing tool. Quality characteristics of the “future work” type are those whose importance is recognized by noticing their deficiency during the evaluation.

3.4.2 Needs for Coverage Measurement in Industry

Although this study showed what coverage criteria were used in the paper, for what purpose, and with what effect, this is just a result of a trial with industrial software, and cannot be said to directly represent the needs of industry. However, it is highly possible that the application of the technologies in industry means that the needs of industry were grasped in the research plan, or the researchers and practitioners in industry developed the technologies to meet the needs of the company, or the seeds met the needs of industry as a result. While it is difficult to grasp that much within a publication, we believe that we are getting closer to the needs by investigating a certain amount.

We consider that improvement of the quality characteristics listed in this study helps the quality of developers’ activities for the purpose of coverage measurement, such as test evaluation and test generation. It may also be enhanced by quality characteristics to the effect of coverage measurements. For example, the higher the usability, the easier it is for developers to understand what tests to add and the higher the defect detectability of the test suite, and the higher the accuracy, the higher the certainty of test sufficiency.

While this is not a fully mature idea, the essential meaning of coverage measurement as required by industry can be read from the survey results as follows: a comprehensive and quick measurement of how well a test suite can find bugs that may occur in a program (even in the future), and feedback to improve the overall efficiency of the development process as test suite improvements, evidence of quality assurance, etc.

3.5 Summary

In this chapter, in order to explore the essential meaning of code coverage measurement in industry, we manually extracted 62 out of 151 automatically collected publications on code coverage measurement in industry, and examined the coverage measurement activities, their purposes, effects, and required quality characteristics described in the publications. The most popular languages for coverage measurement were Java, C, and C++, and the most commonly used coverage criteria were statement/line/block coverage. The most common purpose of coverage was test evaluation, and the most common effect was defect detection, excluding evaluation in experiments. The quality characteristics required for coverage measurements were usability, speed, accuracy, scalability, memory consumption, and reliability. From these results, we concluded that coverage measurement in

industry is required to be comprehensive and quick, and to contribute to the efficiency of the entire development process.

Chapter 4

Virtual Machine for Mutation Analysis

4.1 Overview

Although mutation analysis has been widely studied for several decades, it is rarely used in practical software development flows. The primary reason is a lack of scalability since the analysis generates a large number of mutants. Furthermore, higher order mutation [100], which applies mutation operators more than once, generates an even larger number of mutants.

In this chapter we propose four techniques for high-speed higher order mutation, which are *metamutation*, as a realization of a "do faster" approach, *mutation on virtual machine* as a "do faster" approach, *higher order split-stream execution* as a "do smarter" approach, and *online adaptation technique* as a "do smarter" approach. *Metamutation* technique replaces mutatable program elements with corresponding metamutation functions which can return all possible mutation result in runtime. In compiling target program, metamutation function's call is not modified and optimized out unlike typical (redundant) program elements, and keeps mutation information including the kind of program element and original location of source code. *Mutation on a virtual machine* technique processes metamutated intermediate code on a specialized virtual machine to interprets metamutation function. This does not require a pile of concrete mutated programs but one metamutated intermediate code file, that archives compiling once and invoking process once. *Higher order split-stream execution* branches an execution state into mutated states and a non-mutated state at a point where metamutation function is called on VM, i.e. the corresponding mutatable program element is instructed. Until the point, the mutated execution and non-mutated execution run on a common state, thus running cost is reduced. We extend this technique for higher order mutation. *Online adaptation technique* gives execution states mutation information on the fly, which reduces the number of generated mutants by omitting infeasible mutants.

We had implemented these techniques in our tool MuVM and evaluate it on seven C programs by comparing the number of mutants and an execution time of the whole mutants with an existing higher order mutation tool Milu, existing *Mutant Schemata Generation* (MSG) technique. Additionally, to show the reduction of execution time obtained from applying the split-stream execution, we compare MuVM between with and without split-stream execution. The result of evaluation indicate that MuVM is significantly superior to other tools, such as Milu, MSG, and degraded version of MuVM which excludes split-stream execution.

The main contributions of this chapter are:

- A method to reduce compilation cost by integration of bitcode mutation

and metamutation in C.

- A method to reduce testing time by invoking a process once and splitting execution stream for higher order mutation.
- A method to reduce the number of mutants by an online adaptation technique which omits infeasible mutants.
- An empirical comparison between MuVM and existing techniques.

The rest of this chapter is organized as follows. Section 4.2 provides background on cost reduction techniques for mutation analysis. Section 4.3 introduces our approach for efficient higher order mutation. Section 4.4 describes the design of our tool MuVM. Section 4.5 presents and discusses the results of empirical evaluation. Section 4.6 concludes the chapter and suggests future work.

4.2 Preliminary

4.2.1 Mutant Schemata Generation

Mutant Schemata Generation (MSG) [195] approach aims to reduce the compiling time. Instead of compiling each mutant individually, the mutant schema technique generates a metaprogram which can be used to represent all possible mutants. For example, x + 2 is modified to a metamutation function bo_add(x, 2, ID) where the bo_add function performs one of the arithmetic operators and ID is assigned to each metamutation function respectively in the program. Therefore, to run each mutant against the test suite, only this metaprogram need be compiled, as shown in Fig. 4.1.

The running time of MSG can be expressed as

$$T_{total} = t_{seed} + t_{cmpl} + \sum_{m \in M} t_{inst,m} + \sum_{m \in M} \sum_{tc \in TC} t_{test,m,tc}$$

where $t_{inst,m}$ is the instantiation time with mutant m. Thus, as per this formula, in an MSG approach, the seeding time and the compiling time are reduced to $1/|M|$, and the instantiation time is added, compared to traditional mutation analysis.

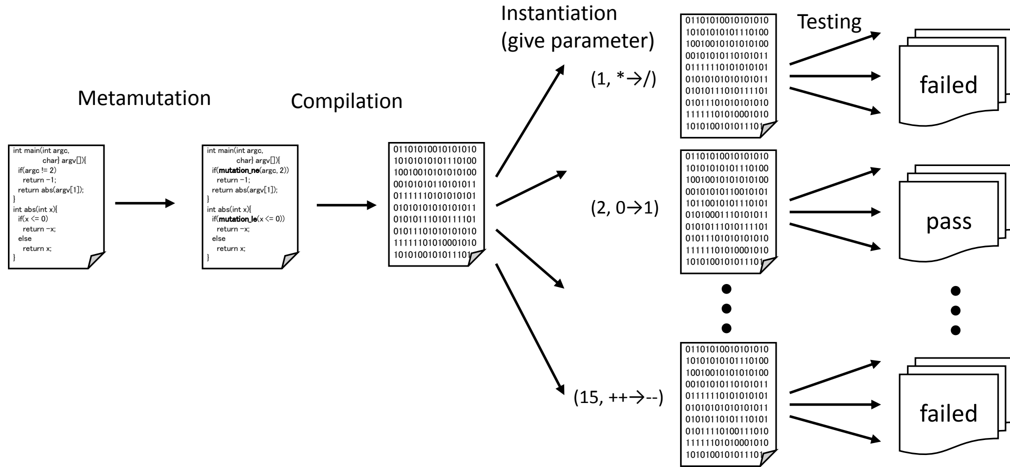

Figure 4.1: Mutation schemata generation

The disadvantage of MSG is an overhead of process forking in test runtime. Some mutants are likely to crash or enter an infinite loop by instantiating the seeded fault thus test processes have to be isolated from the test driver process to monitor them. Assuming the use of unit testing frameworks, the test driver is designed to fork test process for testing mutants. Especially in MSG the test process contains large amount of data for instantiation parameters of all mutants, and in forking the process the data is copied to the test process, which might cause a slowdown.

4.2.2 Bitcode Translation

Bitcode translation, also referred to as "Bytecode Translation Technique" in [130], saves compilation cost by mutating instructions at the bitcode level instead of the source code level. Because mutated bitcode is executed in a virtual machine (VM) directly, we only have to compile the original source code into bitcode once, as shown in Fig. 4.2.

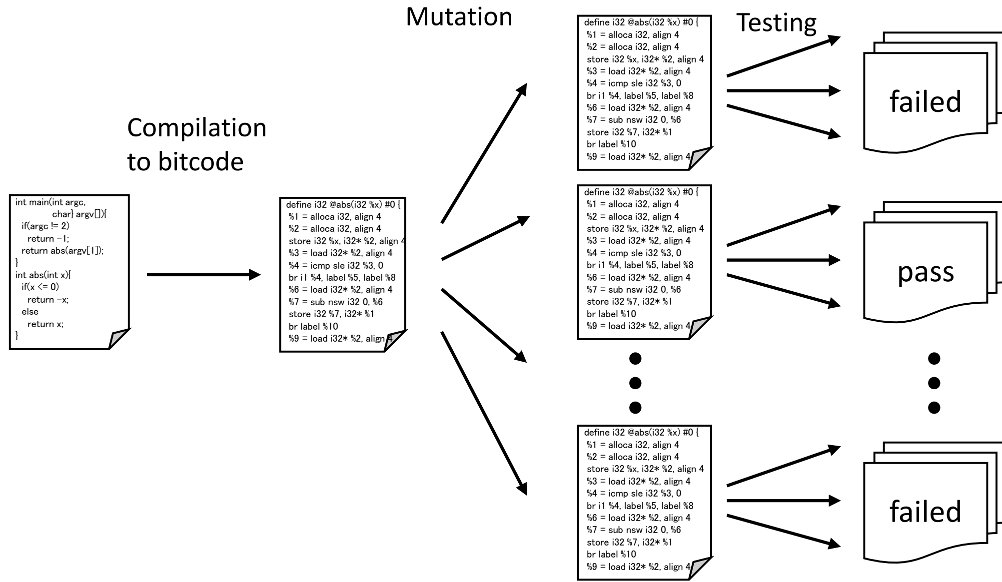

Figure 4.2: Bitcode translation

Therefore the running time of Bytecode Translation Technique is expressed as

$$T_{total} = t_{cmpl} + \sum_{m \in M} t_{seed,m} + \sum_{m \in M} \sum_{tc \in TC} t_{test,m}(tc)$$

which indicates the reduction of compiling time equals $1/|M|$.

This technique can be adapted to only languages which have intermediate representation. LLVM bitcode is a suitable intermediate representation for C programs.

A drawback of this technique is the potential ambiguity in the correspondence between a source code and its bitcode. Take the case of the following two different representations of code

```
1  int func(int x){
2    x = (!x) ? x + 1 : x - 1;
3    return x;
4  }
```

and

```
1 int func(int x){
2   if (x == 0)
3     x++;
4   else
5     x--;
6   return x;
7 }
```

which are semantically equivalent and return the same value according to the input. Applicable mutation operators differ from each other such as “!x” which can be modified by a unary operator mutation and “x == 0” by a binary operator mutation.

These are compiled into almost the same bitcode shown as follows:

```
1 define i32 @func(i32 %x) #0 {
2   %1 = icmp ne i32 %x, 0
3   %.v = select i1 %1, i32 -1, i32 1
4   %2 = add i32 %.v, %x
5   ret i32 %2
6 }
```

Line 2 corresponds to either a unary operation “!x” or a binary operation “x == 0”, but it is not decidable which is the original source code. This indicates that a bitcode-level mutant cannot always be mapped onto its corresponding source-level representation correctly.

Another problem of bitcode translation is mutant omission by optimizer. During compile time, compiler optimizations might delete redundant program elements to reduce the runtime cost and the bitcode size. Thus, not all program elements at source level correspond to bitcode-level elements, and thereby bitcode-level mutations cannot simulate source-level mutations completely. Fig. 4.3 illustrates a case of constant-folding optimization which omits a shift operator. “1 << 7” of original source code is evaluated to 128 in bitcode compile time rather than runtime. Therefore bitcode-level mutation cannot change the shift operator that disappeared due to the optimizer. On the other hand, traditional mutation tool can apply shift operator mutation to the original source code.

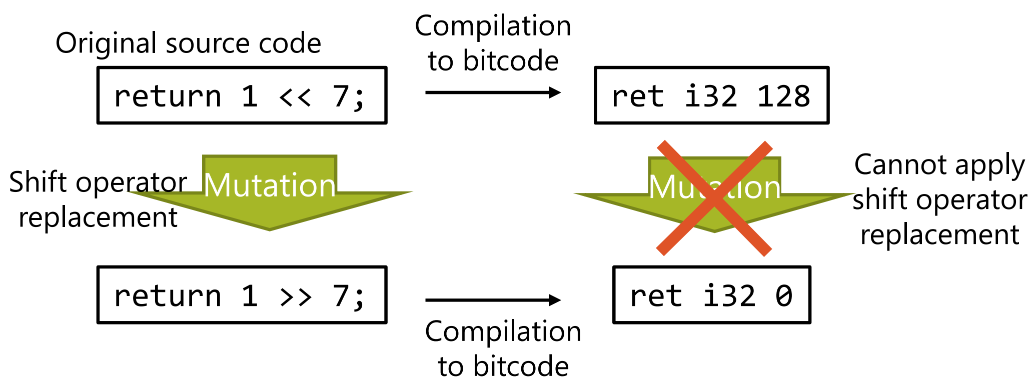

Figure 4.3: Mutants omission by optimizer

4.2.3 Split-stream Execution

King and Offutt [112] initially proposed split-stream execution which splits the execution stream of the original program to begin mutant execution at the point where the mutated statement appears.

This technique shortens testing time by executing common parts together until mutation location is reached as in Fig. 4.4. Hence, if mutation locations are uniformly distributed, the running time is cut down by about half of non split-stream execution in total.

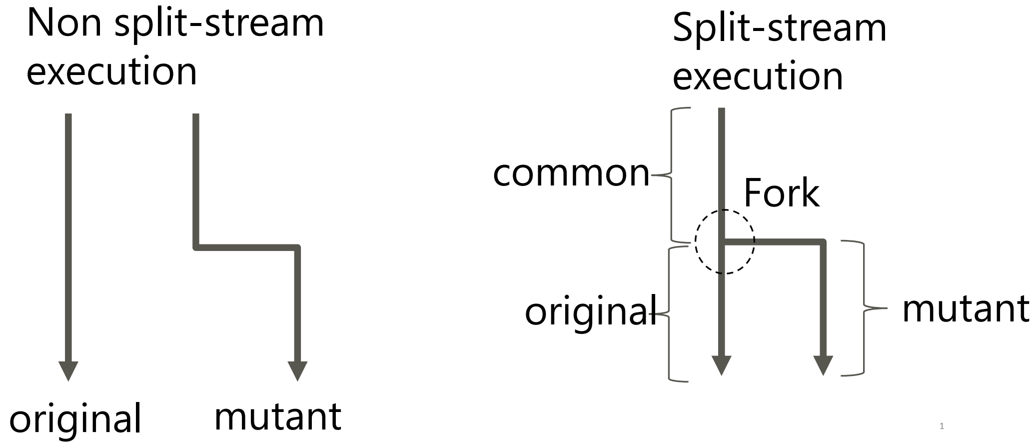

Figure 4.4: Split-stream execution

The limitation of this method is that it is applicable to interpreter-based tool but not to an compiler-based one, because compiler-based tool doesn't manage the execution stream to be split.

4.2.4 Higher Order Mutation

The concept of Higher Order Mutation was proposed by Offutt [156]. In traditional Mutation Testing, mutants can be classified into first order mutants (FOMs) and higher order mutants (HOMs). FOMs are created by applying a mutation operator only once. HOMs are generated by applying mutation operators more than once. HOM could denote subtle faults which are harder to kill than FOM.

Because the combination of mutation operators constructs HOMs, the higher the order of mutants is, the larger the number of feasible HOMs. For instance, if each n mutation location is modified to p kinds of seeded faults in k-th order mutation, the total number of HOMs is $\binom{n}{k} \cdot p^k$. In comparison with the number of FOMs np, this tends to be too enormous to run.

4.3 Techniques

Fig. 4.5 shows an overview of our approach. We adopt four interdependent techniques which compensate for the imperfections of each other. First, the tool replaces program elements by metamutation functions in a way similar to MSG technique. Second, metamutated source code is compiled to bitcode, and third the bitcode is executed with each test case on our tool's VM which splits a execution state into the original instruction and its mutated ones when a metamutation function is called. Finally mutants represented by corresponding mutation descriptors which modify an instruction in the execution state.

4.3.1 Metamutation

In metamutation phase, each mutatable program element, which can be modified by specified mutation operators, is replaced with a metamutation function. Unlike

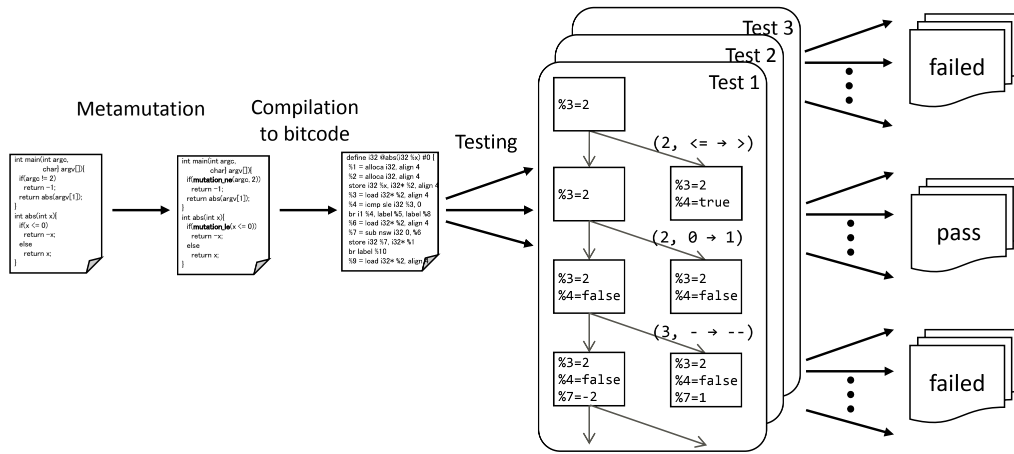

Figure 4.5: Overview of MuVM approach

MSG, our approach does not prepare the native code implementation of the metamutation function. Instead, metamutation functions are activated in the virtual machine.

Fig. 4.6 exhibits an example of a seeded metamutation function. The function signature, including the function name, its arguments and the return type, manifests a type of source-level program element, e.g. "bop_ne(int,int,int)" means an integer typed not-equal ("!=") binary operator. The last argument of the function represents the metamutation ID which is assigned to each metamutation uniquely and is recorded with its mutation location.

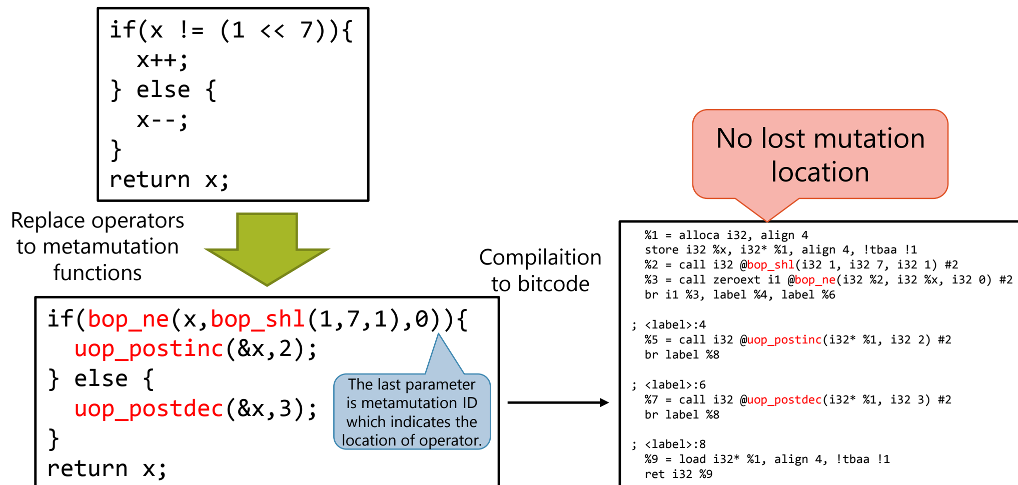

Figure 4.6: Example of metamutation

Metamutation technique is able to compensate for the disadvantages of bitcode-level mutation. The implementation of a metamutation function is deployed in the virtual machine (MuVM) like a built-in function, therefore a compiler cannot see the definition of the metamutation function and cannot decide if this function should be inlined or not. This is the reason that metamutation functions which have constant parameters are preserved during optimization, and all source-level mutants can be reflected at bitcode-level. Also, source-level program elements can be specified at bitcode level by the signature of a metamutation function. Furthermore, the correspondence between source-level and bitcode-level mutation locations is determined by metamutation ID.

4.3.2 Mutation on Virtual Machine

To reduce the compiling and testing time, our tool MuVM is designed as a virtual machine for bitcode-level mutation, which interprets bitcode with seeded metamutation functions and mutates instructions of metamutation functions. If the fetched instruction is a call of a metamutation function, it first checks whether it can be mutated. To decide mutatable or not it checks whether the instruction has been already included in a current trace and whether the number of seeded faults exceeds the maximum mutation order specified by user. The former means that mutation has occurred in only first visiting instructions in order to prevent double counting of mutants. The latter keeps the limit of mutation order. In next step, MuVM branches the execution state into the original state and its mutated states. The detail procedure of branching is explained in the following section. Finally it executes the mutated instructions and the original instruction on each state separately.

This method requires only a one-time compilation similar to bitcode translation technique and reduces the time required to compile a mutant to $1/|M|$. Moreover invoking the tool's process once also reduces testing time, as each test does not create a process but starts the execution state in the tool's process.

MuVM manages the bitcode execution as a virtual machine. Thus it can detect mutant's crash such as buffer overrun and null pointer dereference in running tests. It can also prevent infinite loop by detecting arithmetic overflow and terminating a process by timing out. On these grounds MuVM need not fork the process, this leads the computational costs to lessen.

4.3.3 Higher Order Split-stream Execution

There exist a few mutation analysis tools that adopt split-stream execution. As far as we know, this technique is implemented only in [67] which forks new threads on calling mutated method in Java bytecode.

Our MuVM realizes split-stream execution for higher order mutations (HOSSE). As shown in Fig. 4.7, HOSSE branches not only the original execution states but also their mutated ones to insert additional faults.

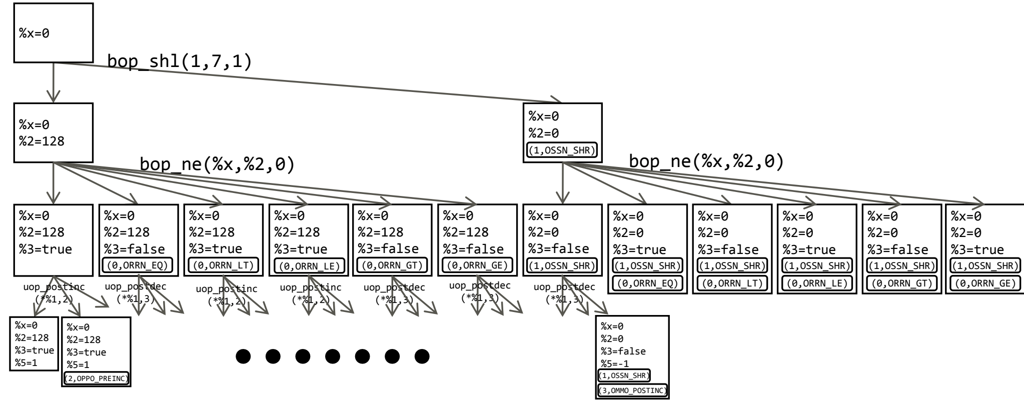

Figure 4.7: Higher order split-stream execution

HOSSE can save the computational cost by sharing the common execution path with a lower-order mutant. We show a brief proof that the computational cost of HOSSE is approximately proportional to the mutation order k. In other words, the higher the mutation order, the faster the HOSSE becomes compared to naive higher order mutations. This is an advantage in considering applications

for higher-order mutations. In addition, to the best of our knowledge, such an analysis has not been done in other SSE or HOM studies.

To analyze the computational cost of HOSSE, we assume an execution sequence as a theoretical model which has n uniformly distributed mutation locations, $\lambda \cdot n$ instructions where λ is the ratio of the number of instructions per mutation location, and p kinds of seeded faults per mutation location in k-th order mutation. We also assume that the execution sequence after mutation has the same execution path as the original execution sequence. Fig. 4.8 (a)–(d) represents the theoretical model at 0, 1, 2, and k-th order mutation, respectively.

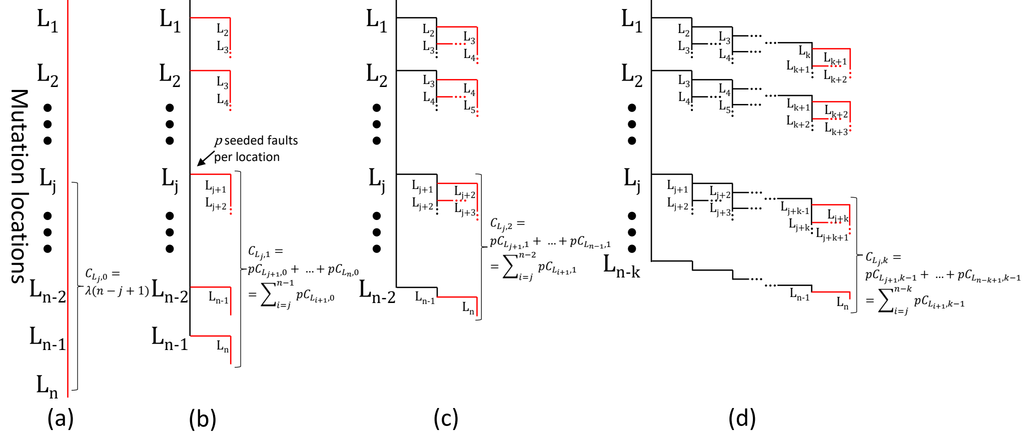

Figure 4.8: HOSSE theoretical model

A partial execution sequence which splits at j-th mutation location, represented by L_j, remains at $n-j+1$ mutation locations and is regarded as $(k-1)$-th order mutation. Therefore total cost of k-th order mutation $c_{L_j,k}$ starting from j-th mutation locations can be expressed as the following recurrence relation.

$$c_{L_j,k} = p \cdot c_{L_{j+1},k-1} + \cdots + p \cdot c_{L_{n-k+1},k-1} = \sum_{i=j}^{n-k} p \cdot c_{L_{i+1},k-1} \tag{4.1}$$

$$c_{L_j,0} = \lambda(n-j+1) \tag{4.2}$$

Note that instructions including more than $(n-k)$-th mutation locations (i.e. $L_{n-k+1}, ..., L_n$) are not available for k-th order mutation. The execution cost up to the branch is not included in Equation 4.1, but is included in the execution cost at the previous order (i.e. $c_{L_j,k-1}$), which indicates that the total cost of HOSSE can be reduced by sharing the execution path.

These equations can be transformed to

$$c_{L_j,k} = \lambda \cdot p^k \binom{n-j+1}{k+1}. \tag{4.3}$$

On the other hand, as we described in 4.2.4, the total number of k-th order mutants is $p^k \binom{n}{k}$, so the computational cost of naive k-th order mutation is $\lambda n \cdot p^k \binom{n}{k}$. Consequently we can get ratio of HOM sequential execution to HOSSE as

$$\frac{cost_{HOM\ seq.\ exec.}}{c_{L_1,k}} = \frac{\lambda n \cdot p^k \binom{n}{k}}{\lambda \cdot p^k \binom{n}{k+1}} = \frac{1}{1-\frac{k}{n}} \cdot (k+1). \tag{4.4}$$

This ratio asymptotically becomes $k+1$ when n is very large, meaning that HOSSE is approximately k times faster than HOM sequential execution when there are a lot of mutation locations.

The detailed proof of the above relation is provided in AppendixA.

4.3.4 Online Adaptation Technique

We classify the generative techniques of mutants into the following two types: offline and online adaptation technique. The offline adaptation technique means each mutant is saved as source code or binary code or intermediate code in the storage or memory, which is exhaustively (i.e. wastefully) generated corresponding to mutatable program elements. These mutants also run on native processes. Existing mutation analysis tools are designed with this technique. On the other hand, the online adaptation technique does not generate code-based mutants but mutation descriptors in execution states, which denotes its metamutation ID and mutation operators derived from. These are called metamutation function and identify the corresponding mutants.

As Fig.4.9, MuVM manages a set of execution states, and adds a mutation descriptor to the execution state when branching the state, namely first calling metamutation function. If the mutation descriptor's metamutation ID is matched with the called metamutation function, the instruction is modified by the mutation operator of the descriptor. That designates no wasteful mutation descriptor (mutant) exists in this method because this method only generates mutation descriptors by actually invoked metamutation function with given test cases.

The online adaptation technique is effective particularly in higher order mutation. In FOM, we can use the traces of the original program with test cases to filter mutatable program elements. In HOM, the program elements which are not in the original program's path might be mutated because the preceding seeded fault may change the path into a different one which includes the program elements the original one cannot execute. That makes the identification of meaningful mutation location difficult. By contrast, our method executes the path switched by the preceding faults. Thus we can easily obtain feasible HOMs and avoid infeasible HOMs.

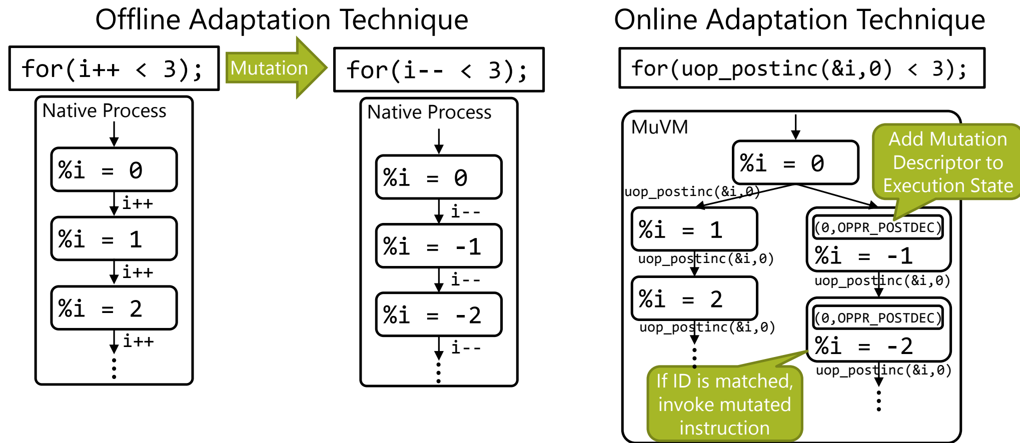

Figure 4.9: State Transition in Offline and Online adaptation Technique

4.4 Design of MuVM

4.4.1 Overall Structure and Behavior

Fig 4.10 shows the structural design of MuVM. MuVM takes a source code, a list of mutation operators and a test suite, and outputs a mutation report which includes the mutation score and mutants. We assume the test suite is written in the format of unit testing framework (Google Test [85]). The mutation operator

list describes the set of mutation operators which the user wants to apply to source code. The mutation report includes the mutation score and other related information like unkilled mutants.

MuVM consists of four main components: Metamutation Inserter, Instruction Mutator, Execution Engine and Mutation Reporter, which act and interact according to the following steps:

1. Metamutation Inserter inserts metamutation functions into the source code and compiles it.
2. MuVM takes a test case from the test suite.
3. Instruction Mutator fetches an instruction from the bitcode.
4. If the instruction is a metamutation function calling which corresponds to a mutation operator in the mutation operator list and the instruction has not been already covered (not in the state's mutants), Instruction Mutator forks the state and adds a mutant to the new state.
5. Execution Engine executes the mutated instruction if the state has a mutant. If not, it executes the instruction as is.
6. MuVM repeats 3)–5) until the state terminates.
7. MuVM runs next state (repeat 3)–6)) if any remaining states exist.
8. If no states, MuVM stores the test results and inputs the next test case.
9. If all test cases are finished, Mutation Reporter outputs a mutation report.

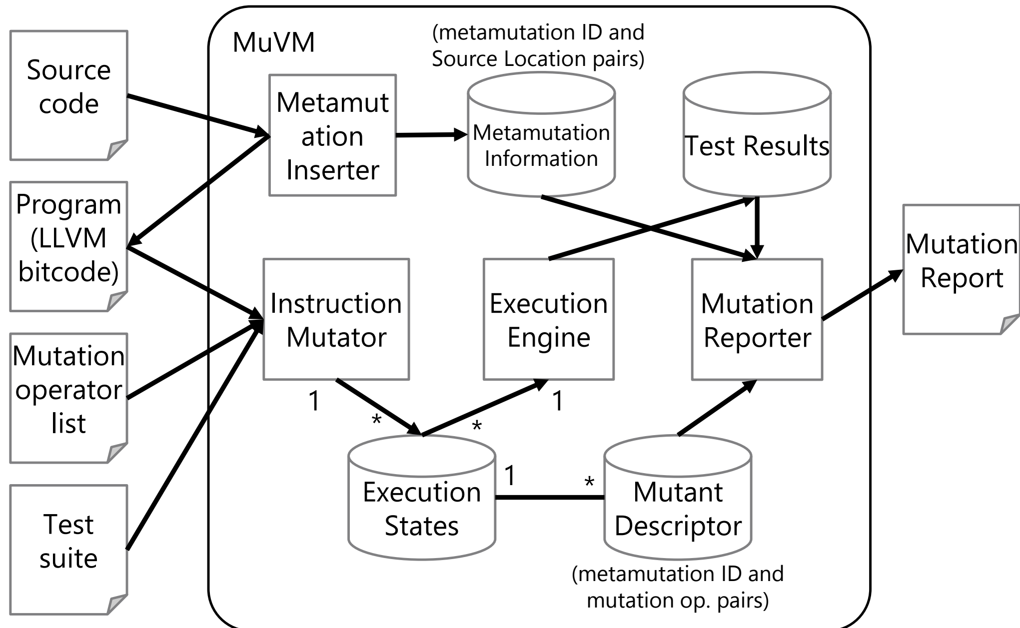

Figure 4.10: Structural design of MuVM

Now we demonstrate an operational example. Suppose an input source code is as in Fig. 4.6. The mutation operator list has relational/shift binary operator replacement (ORRN and OSSN) and increment/decrement replacement (OPPO and OMMO), the test inputs are x = 0, 128 and the expected outputs are 1, 127 respectively.

Given the test case x = 0, MuVM executes the instructions as is until `bop_shl(1,7,1)` is called. Then in calling `bop_shl(1,7,1)` the current execution state forks the original state and its mutated one which has a mutant descriptor (1, OSSN_SHR) which means a change from shift operator "`<<`" into "`>>`" at metamutation ID 1, as Fig. 4.7 shows.

Next, when `bop_ne(%x, %2, 0)` is called, the execution state branches into the original state and 5 mutated ones with mutant descriptors (0, ORRN_EQ), (0, ORRN_LT), (0, ORRN_LE), (0, ORRN_GT) and (0, ORRN_GE) which replace "`!=`" with "`<`", "`<=`", "`>`" and "`>=`" respectively. They can be made unique by combining a metamutation ID and a mutation operator. Then `uop_postinc(*%1, 2)` is processed similarly.

If the original state is terminated at the end of the program, a next state is chosen from the set of derived (mutated) states. Then the execution in the state starts at the point where it branched. Until the termination of the program, it processes the instructions and branches in the same way the original state did except the judgment of mutant's kill. MuVM regards any assertion failure, crashed and timed out mutants as "killed", and mutants which exit normally as "unkilled". These status are stored in the test results.

If all states are finished in a similar manner, MuVM stores the results of each mutant in the test case x = 0, and runs the next test case x = 128 alike. After carrying out all test cases, the tool reports the mutation score calculated from the test results.

4.4.2 Complications

Our approach adopts a metamutation technique like MSG. Therefore we also faced complications of metamutation which Untch [194] pointed out such as (1) short circuit evaluation and (2) structural mutation. Fortunately we were able to apply his solutions to our tool almost similarly.

Short-circuit Evaluation

Short-circuit Evaluation means the semantics of logical connectors where the second operand is not evaluated in case the first operand suffice to determine the value of the expression. For example, the expression `(D != 0 && N/D > 5)` is intended to avoid a "division by zero" error to ignore the right expression `N/D` if the value of `D` is zero, that is, the left expression is evaluated as false. If the logical connector of this expression would be metamutated to `bop_and(D!=0,N/D,`*id*`)`, this evaluates `N/D` before entering the function even if `D` is zero, that might lead to an incorrect mutation.

To evaluate metamutated logical connectors properly, we introduced the following macro.

```
#define bop_and(lhs, rhs, id) \
  (bop_and_(lhs, id) ? rhs : get_lhs(id))
#define bop_or(lhs, rhs, id) \
  (bop_or_(lhs, id) ? get_lhs(id) : rhs)
```

In the original behavior, `bop_and_()` and `bop_or_()` return `lhs` value. In the mutated behavior, `bop_and_()` and `bop_or_()` return negated `lhs` value. Simultaneously `bop_and_()` and `bop_or_()` record `lhs` value with `id` in order to get `lhs` value by calling `get_lhs(id)`, that avoids evaluating `lhs` twice in one metamutation function call. These realize short-circuit evaluation.

Structural Mutation

Some mutation operators change entire statements or the structure of the program. For example, the statement deletion (SDL) operator, which deletes each statements systematically, is one of operators for structural mutations. To metamutate such operators our tool inserts switchable replaced statements shown in Fig. 4.11, where `ssdl(id)` returns true if SDL is enable in specified metamutation ID.

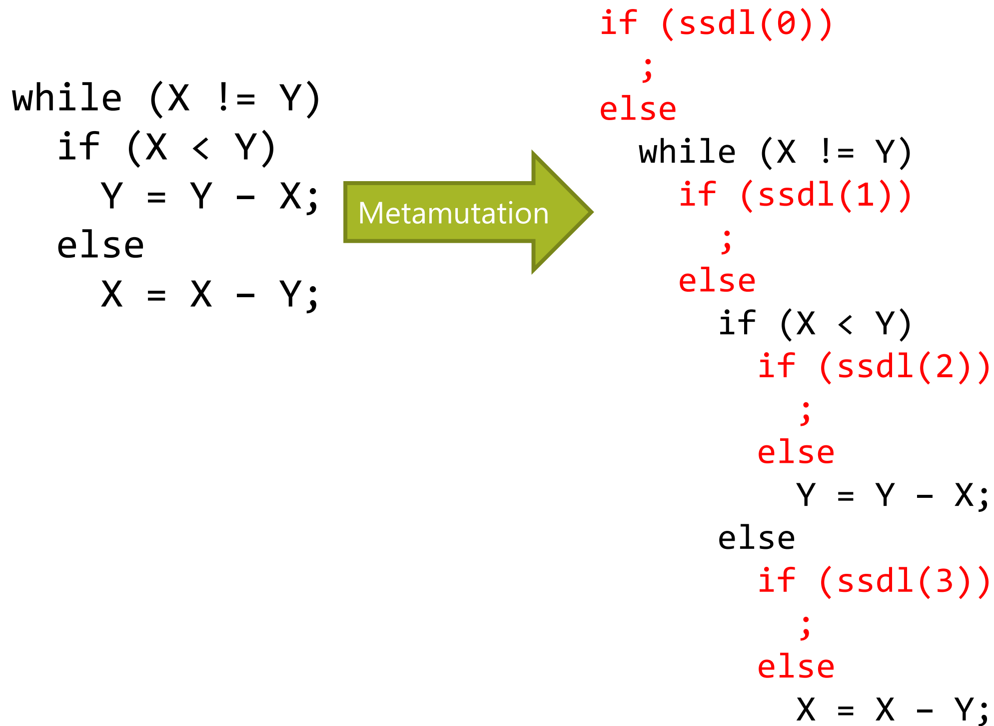

Figure 4.11: Metamutation for Structural Mutation

4.4.3 Mutation Score Calculation

Mutation score without considering equivalent mutants can be expressed by the following form.

$$\frac{(\#\, of\, killed\, mutants)}{(\#\, of\, all\, mutants)}$$

The traditional approach generates and invokes all mutants. Thus the number of all mutants and the number of killed mutants are counted online. On the other hands, our approach omits infeasible HOMs. This means there exists a difference between traditional mutation score and ours. To obtain the same results, our approach needs additional offline processes to offset the omissions to simulate traditional counting. Counting the number of all HOMs is available in offline by simply calculating combination of FOMs, which is referred in 4.2.4. To count the number of killed infeasible HOMs, it is necessary to distinguish killed infeasible HOMs from all infeasible HOMs by means of search for killed feasible HOM which can simulate the killed infeasible HOM. , shown in Figure 4.12.

Here we use HOMs as an example, which is not feasible in MuVM. HOM1 combines the mutant that replaces the comparison operator on line 3 with the

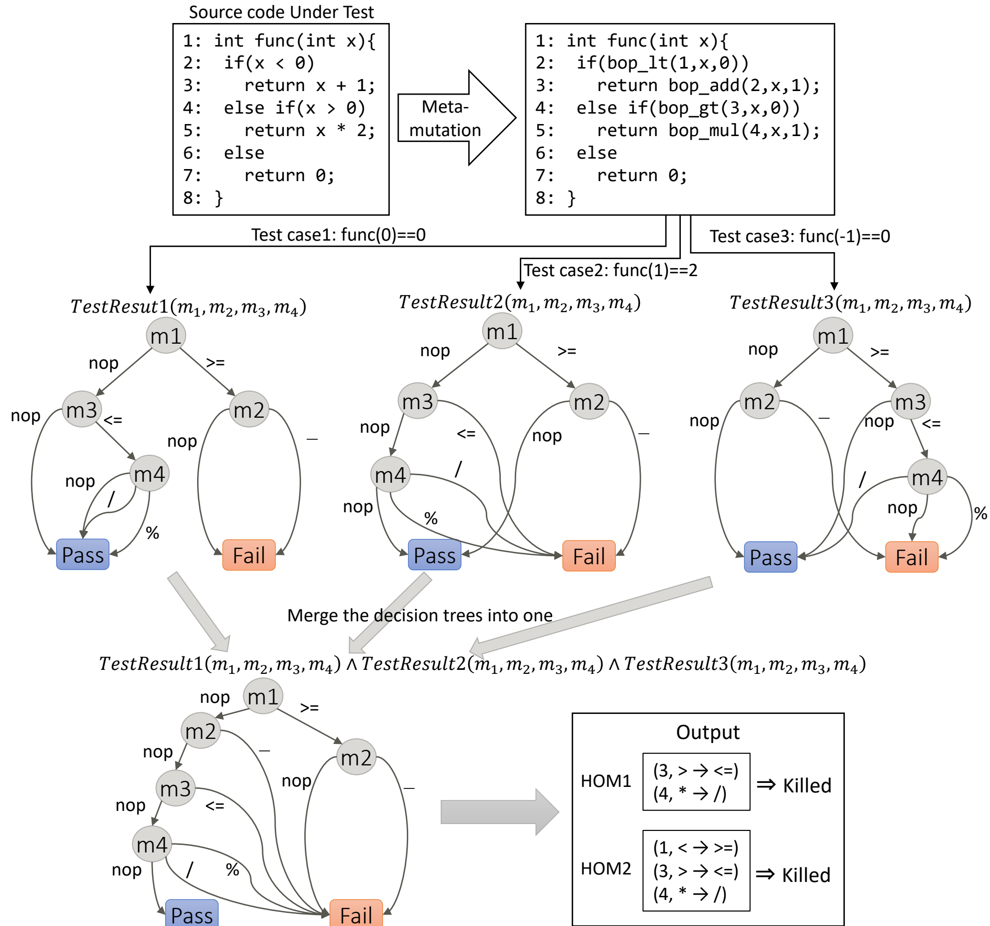

Figure 4.12: Decision tree for simulating infeasible HOM

mutant that replaces the binary operator on line 4. The third line is a return statement, so when the third line is executed, the program terminates there and execution does not reach the fourth line. Such a HOM cannot be generated directly by MuVM. Therefore, we create a decision tree that represents the pass or fail of the test for each combination of mutants, and use it to derive the results of the infeasible HOM. First, we represent the execution result of each test case in MuVM as a decision tree that branches according to the mutation operator applied at each mutation location. If no change is applied, it is represented as a `nop` edge. Next, the decision trees are merged into a single decision tree in the form of a logical product. Finally, the decision tree is used to output the test results for the desired HOM to determine if the HOM can be killed, even if it is infeasible on MuVM. In the case of HOM1 in the example, by following `nop`, `nop`, "<=", we can see that it fails, that is, it can be killed.

Note that this feature has not been incorporated into the implementation of MuVM and has not been used in the following experiments. Specifically, the number of mutants does not include those that are infeasible.

4.5 Evaluation

We investigated the performance of MuVM to answer the following research questions:

Table 4.1: Fundamental data of subject programs

Subject program	Lines of code	# of test cases	# of mutation locations	Description
rand_r	23	1	12	Return random value
ascii_to_bin	16	7	11	Transform a character for cryptography
cal	27	10	14	Compute number of days between input days
tcas	137	1,608	16	Altitude separation
strtol	93	628	22	Convert string to long
space	5,905	100	845	European Space Agency program

RQ1: How many times does our tool invoke mutants compared with fully generative method?

RQ2: How much can our tool reduce computational costs compared with other mutation tools?

RQ3: How superior is our approach to MSG technique?

RQ4: How much does split-stream technique make mutation analysis faster?

4.5.1 Competitive Tools

According to [99] there are some available mutation analysis tools for C. However we could not find any appropriate tool for comparison except Milu [101]. Milu is an open source software and the only tool which can generate HOM in C. Consequently we adopted Milu as a competitor for **RQ1** and **RQ2**. The revision of Milu which we used was the one checked in on Oct 15 2014.

In terms of **RQ3** a tool which supports MSG technique and HOM generation does not exist. Hence we implemented MSG and HOM available tool which utilizes the feature of parameterized test in Google Test [85] to instantiate metamutant by giving sets of mutation descriptors as parameters. This tool was employed for **RQ3**.

Finally **RQ4** requires a degraded version of MuVM which excludes split-stream execution. Instead of forking a state, the degraded MuVM generates a whole new state with the mutant.

4.5.2 Subject Programs

To evaluate the performance of the tools related to $\mathbf{RQ1-4}$ we prepared seven programs written in C. Table 4.1 shows fundamental data of the programs.

We generated test cases for `rand_r` and `ascii_to_bin` by using KLOVER [122] i.e. symbolic execution tool, and employed the prepared test cases for `cal` and `tcas`. The test cases of `space` are sampled in prepared 13,858 test cases. All test cases are formatted to Google Test style in advance.

4.5.3 Experimental Procedure

All experiments except `space` were carried out on a 2.93GHz Intel Xeon X5670 and 4GB of memory with Ubuntu 14.04LTS and Clang compiler. We run the tools

Table 4.2: A Total Number of Invoked Mutants

Subject program	Mutant order	Total # of invoked mutants		Ratio (B / A)
		MuVM(A)	Milu,MSG(B)	
rand_r	1st	48	48	1.00
	1st and 2nd	1,104	1,104	1.00
	1st – 3rd	15,184	15,184	1.00
	1st – 4th	141,904	141,904	1.00
ascii_to_bin	1st	155	350	2.26
	1st and 2nd	1,170	7,945	6.79
	1st – 3rd	5,098	116,375	22.83
	1st – 4th	14,311	1,095,360	76.54
cal	1st	474	610	1.29
	1st and 2nd	10,121	17,870	1.77
	1st – 3rd	131,726	318,130	2.42
tcas	1st	56,366	124,741	2.21
	1st and 2nd	811,805	4,743,316	5.84
strtol	1st	56,582	59,660	1.05
	1st and 2nd	2,092,150	2,756,920	1.32
space	1st	112,255	396,400	3.53
	1st and 2nd	4,951,762	785,527,800	158.64

5 times on each experimental set except parts which did not finish within 100,000 sec. `space`'s experiment requires a huge amount of computational resources and too long execution time to repeat 5 times, thus run once on a 3.5GHz Intel Xeon E3-1275 v3 and 16GB memory with 14.04LTS and Clang compiler. We observed the execution time of each running. Each tool adopts the same mutation operators on the same locations, which are OAAN (arithmetic operator replacement), OAAA (arithmetic assignment replacement) and ORRN (relational operator replacement).

4.5.4 Hypothesis

To answer **RQ2 – 4** explicitly we carried out statistical tests about performance of the tools. We set up null hypothesis as

H_0: there is no difference in performance (execution time) between MuVM and other implementation.

and alternative hypothesis as

H_1: there is a significant difference in performance (execution time) between MuVM and other implementation.

We assume that the two populations have normal distribution with unequal variances. Therefore we employed Welch's t-test for testing $\mathbf{H}_0$ and calculated the t-value and p-value at the 1% significance level. Unfortunately we cannot apply the testing to `space` hence the data of running `space` cannot be taken multiple times.

4.5.5 Results and Discussion

Table 4.3: Execution Time

Subject program	Mutant order	Average of execution time (sec.)				SD of execution time (sec.)				Efficiency (Ave. of Competitor / Ave. of MuVM)		
		MuVM	Milu	MSG	non-SSE	MuVM	Milu	MSG	non-SSE	Milu	MSG	non-SSE
rand_r	1st	2.26	29.77	2.09	2.29	0.02	1.91	0.02	0.01	13.16	0.92	1.01
	1st and 2nd	2.95	598.64	2.65	3.75	0.02	13.08	0.20	0.02	203.20	0.90	1.27
	1st – 3rd	10.92	8,845.97	17.74	23.19	0.09	299.14	0.15	0.10	810.37	1.63	2.12
	1st – 4th	74.89	(82,671.14)	924.44	197.82	0.13	N/A	10.26	0.92	(1103.90)	12.34	2.64
ascii_to_bin	1st	2.35	32.21	2.62	2.44	0.03	1.96	0.07	0.02	13.69	1.07	1.04
	1st and 2nd	2.73	720.11	7.16	3.32	0.02	21.58	0.43	0.02	263.78	2.67	1.22
	1st – 3rd	4.19	10,532.05	270.43	7.04	0.02	376.10	4.63	0.03	2,514.82	64.57	1.68
	1st – 4th	7.49	(99,131.14)	31,162.73	16.08	0.02	N/A	1,424.85	0.09	(13235.13)	4,159.47	2.15
cal	1st	3.07	36.57	14.92	3.67	0.02	0.22	0.04	0.01	11.91	4.86	1.19
	1st and 2nd	13.98	1095.39	375.31	29.66	0.09	12.95	9.86	0.18	78.34	26.84	2.12
	1st – 3rd	134.05	20750.76	9307.54	352.81	1.04	1009.50	286.83	20.03	154.79	69.43	2.63
tcas	1st	89.90	616.61	452.08	232.65	0.37	4.35	20.81	1.78	6.86	5.03	2.59
	1st and 2nd	1,016.62	46,131.82	–	3,498.83	3.72	602.88	N/A	30.79	45.38	N/A	3.44
strtol	1st	152.27	955.31	256.19	280.93	1.41	12.13	8.56	1.75	6.27	1.68	1.84
	1st and 2nd	4,714.18	45,185.22	–	10,192.77	86.15	1,111.61	N/A	61.12	9.58	N/A	2.16
space	1st	8,357.61	5,837.67	–	28,727.10	N/A	N/A	N/A	N/A	0.70	N/A	3.44
	1st and 2nd	64,305.59	(4,321,407.59)	–	–	N/A	N/A	N/A	N/A	(67.20)	N/A	N/A

Table 4.4: The Results of t-test

Subject program	Mutant order	t-value			p-value		
		Milu	MSG	no-SSE	Milu	MSG	no-SSE
rand_r	1st	-45.51	12.74	-2.70	5.98E-12	4.25E-06	3.54E-02
	1st and 2nd	-101.87	3.28	-60.08	5.57E-08	3.05E-02	9.30E-11
	1st – 3rd	-66.04	-85.39	-199.52	3.15E-07	7.95E-12	4.46E-16
	1st – 4th	N/A	-185.22	-296.28	N/A	5.10E-09	7.79E-10
ascii_to_bin	1st	-34.14	-7.65	-5.75	4.39E-06	6.06E-04	4.28E-04
	1st and 2nd	-74.32	-23.59	-40.47	1.96E-07	1.91E-05	1.53E-10
	1st – 3rd	-62.59	-46.07	-190.56	3.90E-07	1.33E-06	6.44E-16
	1st – 4th	N/A	-48.89	-214.80	N/A	1.05E-06	4.15E-11
cal	1st	-334.25	-589.66	-53.70	4.81E-10	1.07E-17	4.23E-08
	1st and 2nd	-186.66	-81.96	-171.61	4.94E-09	1.33E-07	2.64E-12
	1st – 3rd	-45.67	-71.51	-24.39	1.38E-06	2.29E-07	1.68E-05
tcas	1st	-269.89	-38.92	-175.74	1.13E-09	2.60E-06	6.29E-09
	1st and 2nd	-167.33	N/A	-238.87	7.65E-09	N/A	1.84E-09
strtol	1st	-147.08	-26.78	-127.72	1.28E-08	1.16E-05	1.58E-14
	1st and 2nd	-81.17	N/A	-115.98	1.38E-07	N/A	9.34E-13

RQ1: A Total Number of Invoked Mutants

Table 4.2 shows the comparison of total numbers of invoked mutants between our tool and Milu/MSG. The results of `rand_r` indicate no difference between the two. The reason is that `rand_r` has no branch, namely every program element is feasible on any mutant and any test case. On the other hand, there are gaps between the two at other subject programs, and the higher the order of mutants is, the greater is the gap. This is attributed to online adaptation technique which reduces such wasteful HOMs that include any infeasible seeded fault. An infeasible seeded fault wastes the HOMs which consists of combination of faults including this fault. In HOM, the more seeded faults a mutant includes, the more the combination of faults increases, that is, the ratio of mutants which include infeasible seeded faults increases.

RQ2: A Performance Comparison with Other Mutation Tool

Table 4.3 shows the execution times of MuVM and Milu, and Table 4.4 represents the results of t-test for each subject program. The value enclosed in parentheses means estimated value by the number of mutants due to long execution time exceeding 1 day.

In all subject programs at all mutant order except `space`, the p-values are less than 1%. Thus we can reject $\mathbf{H}_0$ and conclude that MuVM is significantly faster than Milu in moderate size of programs.

Milu is implemented using test harness technique, which compiles each mutant into a dynamic library and calls them through the libraries from a test harness. In comparison with compiler-based technique, test harness technique can reduce the test execution cost by creating a process once at every mutant. However each mutant is straightforwardly generated at source level and need to be compiled separately. Based on our investigation of Milu's profile, the compilation is dominant bottleneck over all phase. The ratio of its time reaches over 90% in any program we prepared. By contrast, MuVM needs compilation once and the testing time is dominant rather than compilation time especially in HOM. That seems to make the significant difference between MuVM and Milu in terms of performance.

On the other hands, the result in `space` at 1st order mutation shows that Milu is faster than MuVM. Our observation of `space` found that the `space` program consists of much more loops than other programs. From that we can derive an meaningful aspect of phenomena that the ratio of compilation time to testing time is smaller than other cases. Thus our approach that reduces compilation time is not highly effective in this situation.

As mentioned in section 4.5.5, MuVM tests mutants which have only feasible faults unlike Milu. That also contributes MuVM's faster executions than Milu. Actually there is the correlation between the number of mutants and execution time, where the correlation coefficients are 0.911 in Milu and 0.988 in MuVM.

Other advantage of MuVM is less storage consumption because MuVM does not generate mutated source files to be compiled and the mutated executable files. In `space` at 2nd order mutation Milu tried to occupy 1.5 TB space for 7,855,278 mutated source files, but failed. Our online adaptation approach just requires storage space for metamutated intermediate code and the result of mutation.

RQ3: A Performance Comparison with MSG Technique

Table 4.3 and 4.4 show MuVM is significantly faster than MSG in the majority of higher order mutations. However the 1st order mutation in `rand_r` and the 1st and 2nd order mutation in `rand_r` by MSG are significant superior to MuVM.

By contrast, the 1st and 2nd order mutation in `tcas` and `space` by MSG take a very long execution time and hold a vast amount of memory. Finally the process is killed by OOM killer due to a shortage of memory after 2 days passed.

Our implementation of MSG has a problem about process forking overhead, that is discussed in section 4.2.1. Indeed the tool instantiated enormous number of mutants in HOM, and processes had same number of mutation descriptors. We think that burdened the duplication of the process. This is the reason that the execution times of HOM are extremely longer than FOM.

RQ4: The Effect of Split-Stream Technique

Table 4.3 indicates the execution times of MuVM without split-stream execution for each subject program. Table 4.4 shows that we can reject $\mathbf{H}_0$ in all results except 1st order mutation in `rand_r` owing to the larger p-value than 1%. However, the efficiencies does not have greater gaps such as Milu and MSG. This is the reason that the results depend on only SSE on MuVM whose other features are same as the original version. On the other hand, the higher order the SSE is, the higher the efficiency is. These facts correspond to our analysis of k-th order SSE's efficiency is about $k + 1$ in case of a theoretical model with uniform distribution of seeded faults. Furthermore we should pay the penalty of $k + 1$ times as large as SSE, if the program inevitably needs non-split-stream execution due to external interactions such as file I/O, networking.

4.5.6 Threats to Validity

To compare the tools fairly, we prepared a uniform experimental environment in terms of hardware, operation system, compiler, programing language, subject programs, test suites, and mutation operators. This implies only the difference between the tools' techniques affects the results, and namely that leads to reduction of threats to *internal validity.*

However the collection of subject programs was not randomly chosen, but we selected them in consideration of the diversity, code size, and limitation of the tools. The MSG tool we implemented might not be desirable design in terms of memory usage. Moreover test suites that are not provided by the program authors are generated by a symbolic execution tool. These facts might be threats to *internal validity* and *external validity.*

4.6 Summary

This chapter introduced the techniques which improve the performance of higher order mutation. *Metamutation* replaces mutatable program elements into metamutation function to keep mutation location information in bitcode-level. To reduce compiling time, *mutation on virtual machine* interprets bitcode with seeded metamutation function and mutates instructions of metamutation function. *Higher order split-stream execution* branches original and mutated execution states which represent higher order mutants, and saves execution costs of common parts between each states. *Online adaptation technique* appends a mutation descriptor to an execution state to omit infeasible mutants. Our comparative

experiments demonstrated that our tool significantly is better than the existing one, existing technique (mutation schema generation), and no-split-stream execution in higher order mutation.

Our tool has limitation about compiler optimization. Although seeding metamutation function prevents mutatable program elements from elimination by the optimizer, it gives up gaining benefits from optimization. Less optimized program needs to pay extra execution cost than well optimized one.

MuVM is able to generate feasible HOMs exhaustively in a reasonable time. On the other hand, the underlying motivation of HOM is to seek and find rare but valuable mutants such as subsuming HOMs [100] which is harder to kill than FOMs from which it is constructed. Our tool can provide only the population of subsuming HOMs rapidly. An effective search for subsuming HOMs can be an interesting future work.

Chapter 5

Statement Deletion Mutation-based Fault Localization

5.1 Overview

In this chapter, we propose a method using the statement deletion mutation operator for mutation-based fault localization (SDL-MBFL). The evaluation uses actual source code, test cases, and faults from an enterprise system re-engineering project. Most evaluations in existing fault localization research use OSS. Evaluations using industrial software are very limited. In particular, to the best of our knowledge, no industrial evaluation of mutation-based fault localization exists. An intrinsic characteristic of the subject project is that it is necessary to identify multiple faults simultaneously. This is because what the term "fault" in the present context refers to incompatibilities before and after re-engineering, and because there are incompatibilities that should not be fixed such as specification improvements, it is impossible to identify and fix them one by one. As a result of the evaluation, the execution time of SDL-MBFL was reduced by 20.3% compared with that of MBFL, and the identification of fault location within the 100th place of suspiciousness was superior to that of other fault localization methods. This suggests that SDL-MBFL may have excellent fault localization performance for multiple faults while keeping the execution cost low.

In summary, the contributions of this study are as follows.

- We propose and implement mutation-based fault localization using statement deletion mutation.
- We compare fault localization techniques, including SDL-MBFL, with real-world software and several faults, in an enterprise context.
- Compared with conventional MBFL, SDL-MBFL has the same number of detected faults and can reduce the execution time by 20.3%.

The structure of this chapter is as follows. As a background of this study, fault localization based on statement deletion mutation is introduced in Section 5.2. Section 5.3 describes research questions and evaluation methods. Section 5.4 presents the results of the evaluation. Section 5.5 discusses our method and evaluation.

5.2 Preliminary

This section introduces statement deletion mutation analysis and general fault localization and explains fault localization based on statement deletion mutation.

We also introduce a hybrid approach that combines multiple fault localization techniques.

5.2.1 Statement Deletion Mutation

The statement deletion mutation operator (SDL), as the name suggests, is a mutation operator that corresponds to deleting one or more statements from the program under test. An example is shown in Figure 5.1. In the example, three statement deletion mutations are applied to the program under test, generating mutants that delete the entire `if` command on lines 2-4, the return statement on line 3, and the return statement on line 5, respectively. Deng et al. [56] and Delamaro et al. [51] experimentally demonstrated that all operators can be used exclusively in SDL.

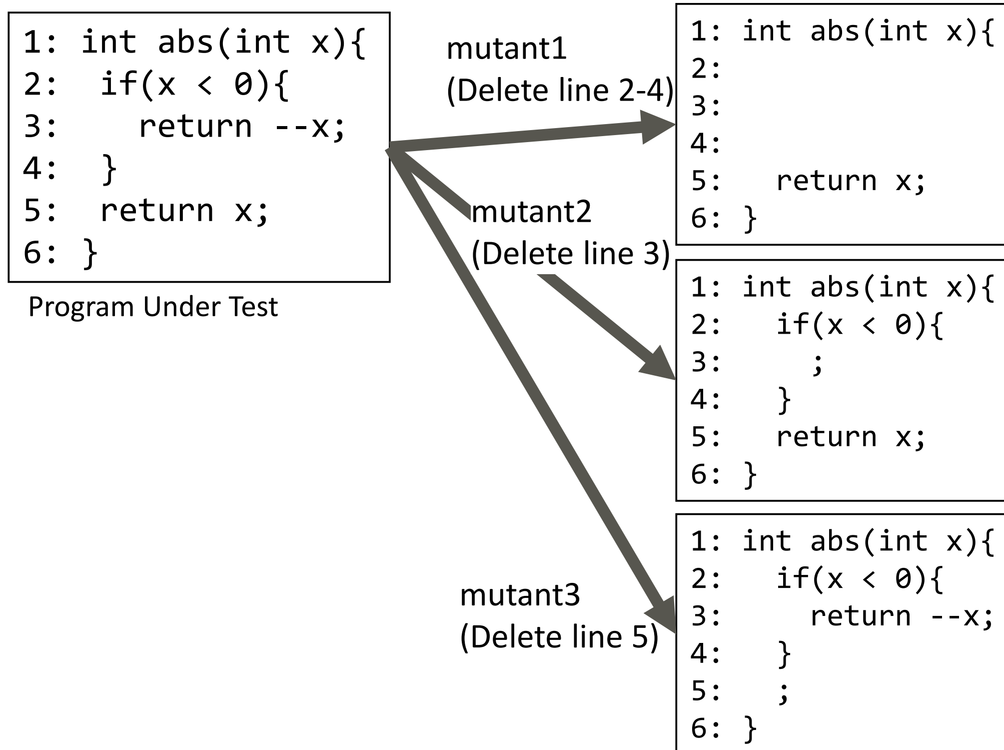

Figure 5.1: Statement deletion mutation

5.2.2 Spectrum-based Fault Localization

SBFL is a technique to measure the suspiciousness of failure-causing points by statistically treating the statement coverage information (executable statement spectrum) in passing and failing test cases. This technique counts the number of executed passed and failed test cases for each statement in the program, and the statements with more executed failed test cases and fewer executed pass test cases have higher suspiciousness. Conversely, statements with fewer failed test cases executed and more passed test cases executed are given low suspiciousness. Various formulae have been proposed to calculate the suspiciousness $Susp(s)$ of a given statement s. The following are some representative examples:

$$Tarantula : Susp(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed} + \frac{passed(s)}{totalpassed}} \tag{5.1}$$

$$Ochiai : Susp(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}} \tag{5.2}$$

$$Op2 : Susp(s) = failed(s) - \frac{passed(s)}{totalpassed + 1} \tag{5.3}$$

$$DStar : Susp(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))} \tag{5.4}$$

where $passed(s)$ is the number of passed test cases that executed statement s, $failed(s)$ is the number of failed test cases that executed statement s, $totalpassed$ is the number of all passed test cases, and $totalfailed$ is the number of all failed test cases. DStar's $*$ is a variable representing a positive real number, which is assumed to be 2 in this chapter.

5.2.3 Mutation-based Fault Localization

MBFL uses not only trace information but also the result of mutation analysis to find the fault location, using the pass or fail status of tests, as in SBFL. In general mutation analysis, we assume that there are no test cases that fail in the original program and measure the fault detectability of the test suite by examining test failures when it is run against mutants, i.e., whether it is able to detect the fault. MBFL executes mutants with test cases that fail in the original program and ranks the fault locations based on the reasoning that the greater the influence of mutants on the output of the failed test case, the greater the suspiciousness at the mutation location. Therefore, simply finding a test case that can kill a mutant, as in general mutation analysis, is not sufficient as we require information on whether the mutant could be killed by each test case. Note that there may be more than one mutant per statement.

Representative techniques for MBFL are MUSE [141] and Metallaxis [165].

MUSE calculates the suspiciousness in the mutant m_i using the following formula:

$$Susp(m_i) = failed(m_i) - \frac{f2p}{p2f} \cdot passed(m_i) \tag{5.5}$$

where $failed(m_i)$ is the number of failed test cases in the original program that changed from failed to passed by mutation m_i, and $passed(m_i)$ is the opposite, i.e., the number of passed test cases in the original program that changed from passed to failed by mutation m_i. $f2p$ is the total number of failed test cases in the original program that changed from failed to passed by arbitrary mutation, and $p2f$ is the opposite, i.e., the total number of passed test cases in the original program that changed from passed to failed by arbitrary mutation. To change the per-mutant suspiciousness to per-statement suspiciousness, let $Susp(s) = Avg_{m_i \in mut(s)} Susp(m_i)$, where $mut(s)$ is the set of mutants in statement s.

Metallaxis calculates the suspiciousness in the mutant m_i using the following formula based on Ochiai's equation:

$$Susp(m_i) = \frac{failed(m_i)}{\sqrt{totalfailed \cdot (failed(m_i) + passed(m_i))}} \tag{5.6}$$

In this study, in addition to the Ochiai-based formula 5.6, the following formulas based on Tarantula, Op2, and DStar are also used:

$$Tarantula : Susp(m_i) = \frac{\frac{failed(m_i)}{totalfailed}}{\frac{failed(m_i)}{totalfailed} + \frac{passed(m_i)}{totalpassed}} \tag{5.7}$$

$$Op2 : Susp(m_i) = failed(m_i) - \frac{passed(m_i)}{totalpassed+1} \quad (5.8)$$

$$DStar : Susp(m_i) = \frac{failed(m_i)^*}{passed(m_i)+(totalfailed-failed(m_i))} \quad (5.9)$$

where $failed(m_i)$ is the number of failed test cases in the original program where the output of the mutant m_i and the original program are different, $passed(m_i)$ is the number of passed test cases in the original program where the output of the mutant m_i and the original program are different, $totalfailed$ is the number of all failed test cases in the original program, and $totalpassed$ is the number of all passed test cases in the original program.

The difference between MUSE and Metallaxis is the extent to which they check the impact of mutations on the output; MUSE raises the suspiciousness of mutations that make a failing test case pass, i.e., the mutations that make the output match the expected result. By contrast, Metallaxis raises the suspiciousness of a mutation merely if the output of the failed test case is changed by the mutation, even if this changed output does not match the expected result. Because of these differences, the general trend in comparing the two is that MUSE has more false negatives and, conversely, Metallaxis has more false positives. Previous evaluations comparing Metallaxis and MUSE [165, 169] have shown Metallaxis to be superior. Therefore, we use Metallaxis as the underlying technique to calculate suspiciousness in MBFL, for the purpose of this study.

5.2.4 Statement Deletion Mutation-based Fault Localization

SDL-MBFL is a method using only the SDL mutation in mutation-based fault localization. In general MBFL, there can be multiple mutations in a single instruction, and depending on the choice of mutation operator, there can be statements that do not have any mutations at all, which affects the execution cost and fault localization performance. Therefore, by using SDL, we realized an MBFL that always performs one mutation per statement.

Figure 5.2 and Figure 5.3 illustrate fault localization using SBML and SDL-MBFL respectively, on the same motivating example. The function under test, `mid()`, is expected to return the median value among the three arguments `x`, `y`, and `z`. Statements s_2 and s_7 contain faults, where the correct versions `m = z;` and `m = x;` have been mistakenly replaced with `m = x;` and `m = y;` respectively. This results in the failure of tests `(5,3,4)` and `(2,1,3)` out of the six test cases. In Figure 5.2, the lines executed in each test case are indicated by •. In Figure 5.3, there is a statement deletion mutant corresponding to each line, where $\surd$ represents the mutant that changes the output to be different from the output of the original program.

SDL-MBFL is based on Metallaxis and Tarantula's formula for calculating suspiciousness. In Figure 5.3, the ranks of suspicious values for the mutation locations where s_2 and s_7 are deleted is 4th and 2nd, respectively, which are higher than the rank in SBFL. This is because each statement is comprehensively mutated by the SDL, and by measuring the impact of each statement as it changes, the localization is successful even for statements that are executed by a large number of passed test cases. This also shows that it is excellent in terms of efficiency, as there is only one mutation per statement.

Furthermore, one of the advantages of MBFL in general, as well as SDL-MBFL in particular, is that it can be effective when multiple faults are simultaneously present. For SBFL, because the suspiciousness is calculated based only on the

aggregate number of passed/failed test cases, it is likely to be affected by test cases that fail due to other faults. By contrast, in the case of MBFL, the suspiciousness of a statement is less affected by other faults, since the effect on the output is calculated independently for each mutant. Therefore, MBFL is considered better than SBFL at localizing multiple faults [141].

	Program Under Test	Test cases and the traces						#passed	#failed	SBFL (Tarantula) susp.	rank
	`int mid(int x, int y, int z){`	3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 3				
S_1	`int m;`	●	●	●	●	●	●	4	2	0.500	7
S_2	`m = x; //m = z;`	●	●	●	●	●	●	4	2	0.500	7
S_3	`if (y < z)`	●	●	●	●	●	●	4	2	0.500	7
S_4	`if (x < y)`	●	●			●	●	2	2	0.667	3
S_5	`m = y;`		●					1	0	0.000	13
S_6	`else if (x < z)`	●				●	●	1	2	0.800	1
S_7	`m = y; //m = x;`	●					●	1	1	0.667	3
S_8	`else`			●	●			2	0	0.000	13
S_9	`if (x > y)`			●	●			2	0	0.000	13
S_{10}	`m = y;`			●				1	0	0.000	13
S_{11}	`else if (x > z)`				●			1	0	0.000	13
S_{12}	`m = x;`							0	0	0.000	13
S_{13}	`return m;`	●	●	●	●	●	●	4	2	0.500	7
	`}`	P	P	P	P	F	F				

Figure 5.2: Spectrum-based Fault Localization

5.2.5 MBFL and SBFL Hybrid Approach

Pearson et al. [169] proposed the following hybrid methods combining MBFL and SBFL.

- Hybrid-Failover: Suspiciousness for non-mutable statements using SBFL values
- Hybrid-Average: Average the MBFL and SBFL suspiciousness
- Hybrid-Max: The MBFL and SBFL results are compared and the larger is adopted.

Pearson et al. found the Hybrid-Average approach to be the most effective, in their experiments.

	Program Under Test	Test cases and mutants which change the output						Deleted statement	#passed	#failed	SDL-MBFL susp.	rank
	`int mid(int x, int y, int z){`	3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 3					
S_1	`int m;`							–	0	0	0.000	13
S_2	`m = x; //m = z;`				✓	✓		S_2	1	1	0.667	4
S_3	`if (y < z)`		✓	✓			✓	S_3-S_{12}	2	1	0.500	6
S_4	`if (x < y)`		✓				✓	S_4-S_7	1	1	0.667	4
S_5	`m = y;`		✓					S_5	1	0	0.000	13
S_6	`else if (x < z)`						✓	S_6-S_7	0	1	1.000	2
S_7	`m = y; //m = x;`						✓	S_7	0	1	1.000	2
S_8	`else`			✓				S_8-S_{12}	0	0	0.000	13
S_9	`if (x > y)`			✓				S_9-S_{12}	1	0	0.000	13
S_{10}	`m = y;`			✓				S_{10}	1	0	0.000	13
S_{11}	`else if (x > z)`							S_{11}-S_{12}	0	0	0.000	13
S_{12}	`m = x;`							S_{12}	0	0	0.000	13
S_{13}	`return m;`	✓	✓	✓	✓	✓	✓	S_{13}	4	2	0.500	6
	`}`	P	P	P	P	F	F					

Figure 5.3: Statement Deletion Mutation-based Fault Localization

5.3 Evaluation Setup

This section describes how we evaluate the fault localization techniques.

5.3.1 Research Questions

The following research questions were addressed in this study:

- RQ1: How long does each mutation analysis run?
- RQ2: What is a good formula for calculating the suspiciousness of SDL-MBFL?
- RQ3: Does SDL-MBFL rank high in faults compared to other fault localization methods?
- RQ4: Does the hybrid method of SDL-MBFL and SBFL rank high in faults?

For RQ1, we investigate how fast the execution speed of SDL-MBFL is compared to that of other MBFLs. For RQ2, we investigate which of the formulas shown in 5.6 - 5.9 can rank the real faults high when using SDL-MBFL. For RQ3, we measure the fault localization performances of SBFL, MBFL, and MBFL without SDL and investigate whether SDL-MBFL ranks higher in real faults than other fault localization techniques. For RQ4, we measure the fault localization performance of the hybrid method of SDL-MBFL and SBFL and explore the best way to combine them.

5.3.2 Tool

Based on MuVM shown in Chapter 4, we implemented a mutation-based fault localization tool in the C programming language. As described in Chapter 4, MuVM is a tool that realizes high-speed mutation analysis by the following features.

- Metamutation
- Mutation on Virtual Machine
- Higher Order Split-stream Execution
- Online Adaptation

5.3.3 Evaluation Subjects

Fujitsu, a Japanese multinational company providing IT systems and services, has developed a hardware monitoring system for server products and storage systems, which flags hardware malfunctions through e-mail alerts. The programs of the monitor are individually developed. Consequently they are different source codes and their SMTP libraries have never been the same. However the SMTP libraries have similar features such as sending mails on SMTP, SMTP AUTH, POP before SMTP, S/MIME and mail fragmentation. So Fujitsu decided to re-engineer the SMTP library to unify the server products and the storage systems shown in Figure 5.4. They aim to reduce maintenance costs and add new features. During this process we run into the problem of comprehensively testing the compatibility of the old and new systems.

To check for incompatibilities introduced by the system re-engineering, we apply the symbolic execution tool KLEE to the server program before the migration and then run the generated test cases against the version after the migration.

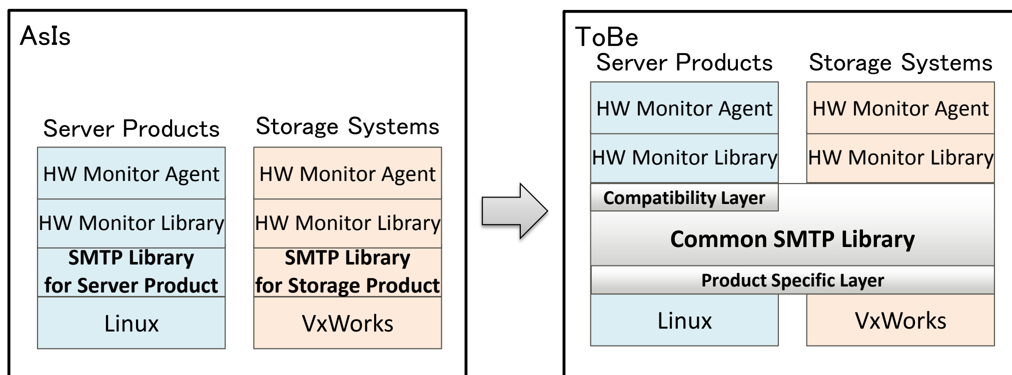

Figure 5.4: Overview of system re-engineering project

A summary of the post-migration programs and tests is given in Table 5.1. Testing is done on the entire library through a single API, not on a per-function or per-file basis.

Table 5.2 lists the subject faults to be identified by the fault localization technique. Although we use the expression "fault" here for convenience, it is actually an incompatibility before and after the migration, and not all of the incompatibilities can be corrected because there are acceptable incompatibilities. Moreover, while these faults have been validated by the developers of the subject system, it is possible that they are not all the faults present because how they can be fixed has not been disclosed.

Table 5.1: Overview of subject program

Usage	SMTP Library for Server Monitors
Languages	C
Size	13 KLOC
Executable	5496 LOC
Total Test Cases	10876
# of Failed Test Cases	4003
Statement Coverage	86.3%
Faults	9

Table 5.2: Subject faults

Fault ID	File	Line No.	Error type
1	dir_c/src01.c	524-535	condition error
2	dir_c/src01.c	443,445	omitted statements
3	dir_c/src01.c	507-508	redundancy statements
4	dir_r/src07.c	292	condition error
5	dir_r/src03.c	309,311,312	processing sequence error
6	dir_r/src06.c	216	omitted statements
7	dir_r/src06.c	183,184	logic error
8	dir_r/src07.c	216-234	redundancy statements
9	dir_r/src10.c	266	macro error

5.3.4 Evaluation metrics

Fault localization techniques can rank all statements by ordering them in descending order of suspiciousness. However, evaluating the performance of a fault localizaton method across programs of different sizes, in a uniform manner, presents a particular challenge. Metrics such as EXAM score [202], LIL [141], T-Score [35], and Expense [104] have been proposed for this purpose. The widely used EXAM score is obtained by $\frac{n}{N}$, where n is the rank of suspicious statements and N is the total number of statements. This metric tries to capture the percentage of the total statements that would need to be examined by the user, to find the faulty one, when going down the ranked list of suspicious statements provided by fault localization, one statement at a time. These evaluation metrics fall short in two scenarios, namely when multiple statements receive the same suspiciousness score, and in the case of multi-statement faults.

Multiple statements with the same score

If an element has the same suspiciousness score as another candidate element, the ranking is the same. In this case, we use another evaluation metric $E_{inspect}$ [211], although we often use the average of the rankings. $E_{inspect}$ is designed to solve the problem of rank averaging. For example, if all the elements with the same score are faults, the average rank is unreasonably low even though the user can check any of them early.

Assuming that the number of elements having the same score is n, the number

of faulty elements having the same score is n_f, and the starting position of the elements having the same score is P_{start}, $E_{inspect}$, which is defined as follows:

$$E_{inspect} = P_{start} + \sum_{k=1}^{n-n_f} k \frac{\binom{n-k-1}{n_f-1}}{\binom{n}{n_f}}$$

This formula calculates the probability of the first element appearing in the kth location starting from P_{start}, the numerator part of the which $\binom{n-k-1}{n_f-1}$ is the number of all combinations where the first faulty element is at k, and denominator part $\binom{n}{n_f}$ is the number of all combinations.

Table 5.3 shows an example of suspiciousness ranking to further understand difference between $E_{inspect}$ and the average of the rank. The elements in the table are sorted in descending order of suspiciousness score. The element IDs B, C, D and E have the same suspicious score, so they should be ranked the same, although the written rank is different. The average ranking becomes $(2+3+4+5)/4 = 3.5$, while for $E_{inspect}$ it becomes $2 + 1 \cdot \binom{2}{2}/\binom{4}{3} = 2.25$. Assuming that we check whether the elements are faulty or not in order from top to bottom, the expectations of the rank in which faulty elements can be found are higher than the average rank since the number of faulty elements in the same rank is greater than the number of all elements in the same rank. In this kind of case, $E_{inspect}$ is considered to be more consistent with the actual debugging behavior.

Table 5.3: Example of suspiciousness ranking for calculating $E_{inspect}$

Rank	Element ID	Susp.	Faulty?
1	A	1.0	No
2	B	0.8	Yes
3	C	0.8	Yes
4	D	0.8	No
5	E	0.8	Yes

Note that $E_{inspect}@n$ represents the number of faults in the top n when ranked by $E_{inspect}$. For example, $E_{inspect}@3$ in Table 5.3 is 3 because top 3 elements include elements ID B, C and E with $E_{inspect}$ of 2.25.

Multi-statement faults

Multiple statements of faults may have different ranks on each statement. [169] proposed an evaluation method for the following three scenarios:

1. Best-case: one of the faulty statements needs to be identified

2. Worst-Case: All faulty statements need to be identified

3. Average-Case: 50% of faulty statements need to be identified

Note that all three scenarios simplify to the same identical case when there is only one faulty statement. In this thesis, we adopt the best-case scenario unless otherwise stated.

5.4 Evaluation Results

We show the results of evaluation of each RQ based on the methods presented in Section 5.3.

Table 5.4: Results of mutation analysis

	SDL-MBFL	MBFL	MBFL w/o SDL
Duration (hours)	150.8	189.3	38.5
# of mutants	2,734	4,172	1,438
Total number of test cases executed	5,016,143	8,089,454	3,073,311
Mutation score	22.17%	26.01%	33.31%
Mutation operators	SSDL	OAAN, OBBN, ORRN, OSSN, OAAA, OBBA, OSSA, SSDL	OAAN, OBBN, ORRN, OSSN, OAAA, OBBA, OSSA

5.4.1 RQ1: How long does each mutation analysis run?

Table 5.4 shows the results of MBFL using only SDL (SDL-MBFL), MBFL using common mutation operators including SDL, and MBFL using mutation operators excluding SDL (MBFL w/o SDL). In the table, we describe the statement deletion mutation operator as "SSDL,'' not "SDL,'' in accordance with the notation in [3].

The execution time of SDL-MBFL is 20.3% less than that of MBFL. This is because the number of executed mutants is as high as 65.5% of the number of mutants in MBFL, although only one type of mutation operator is used by SDL-MBFL.

5.4.2 RQ2: What is a good formula for calculating the suspiciousness of SDL-MBFL?

Figure 5.5 and Table 5.5 show the results of SDL-MBFL for each suspiciousness formula.

Among them, Tarantula was the best in both $E_{inspect}$ and EXAM scores. There was almost no difference in $E_{inspect}$ and EXAM scores between Ochiai and DStar, and Op2 was the worst.

5.4.3 RQ3: Does SDL-MBFL rank high in faults compared to other fault localization methods?

A comparison of SDL-MBFL with other fault localization techniques is shown in Figures 5.6 and 5.4.

SDL-MBFL found more faults in the 100th position than SBFL or MBFL without SDL. However, MBFL is equal to or slightly inferior to SDL-MBFL, indicating that the inclusion of SDL in the mutation operator has a dominant influence on the fault localization performance. EXAM scores are better for SBFL and MBFL without SDL.

5.4.4 RQ4: Does the hybrid method of SDL-MBFL and SBFL rank high in faults?

The results of comparing the hybrid method of SDL-MBFL and SBFL are shown in Figure 5.7 and Table 5.4.

For $E_{inspect}$@20, we found that SDL-MBFL is superior to other hybrid methods. However, Hybrid-Average and Hybrid-Max are superior in terms of $E_{inspect}$@100 and EXAM scores. Overall, Hybrid-Max is better than the other hybrid approaches.

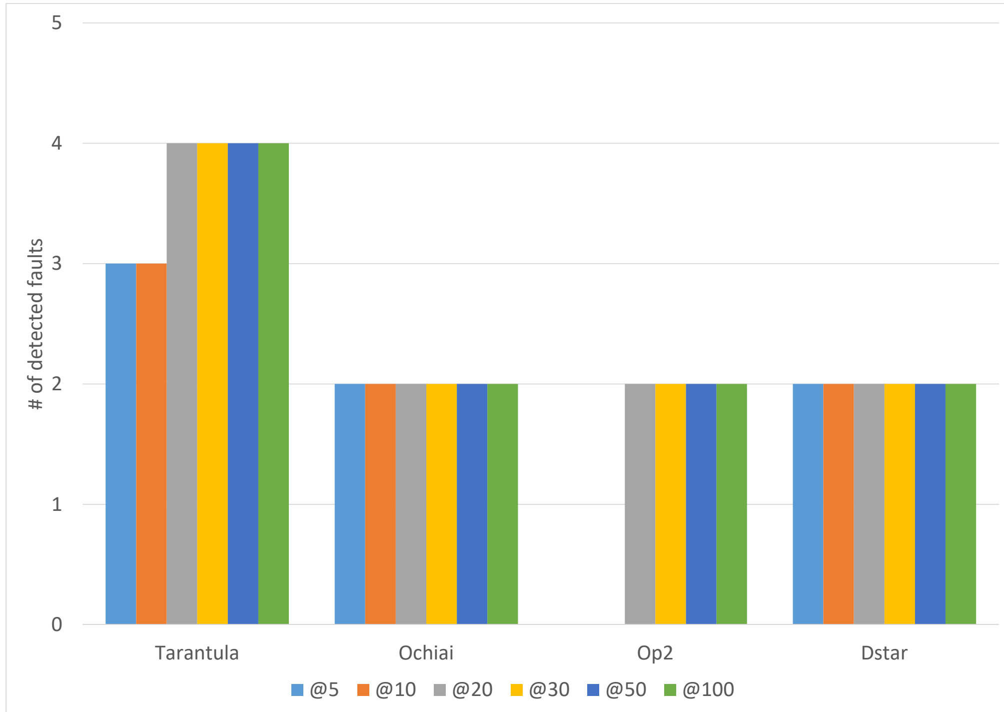

Figure 5.5: $E_{inspect}@n$ in each of the SDL-MBFL suspiciousness calculation formulas

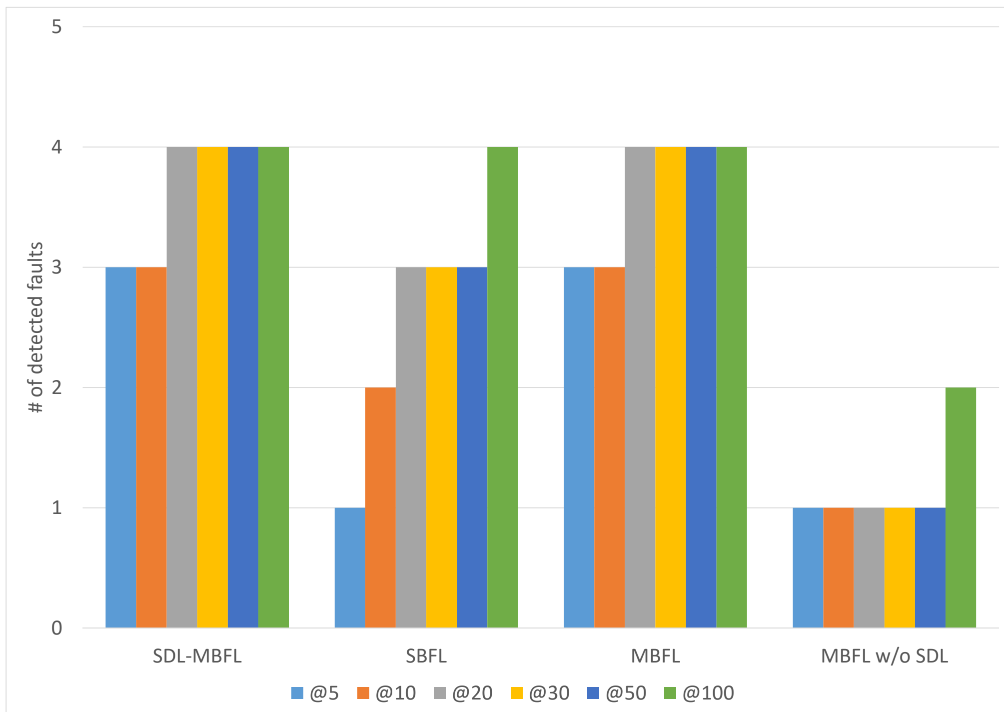

Figure 5.6: $E_{inspect}@n$ for each fault localization technique

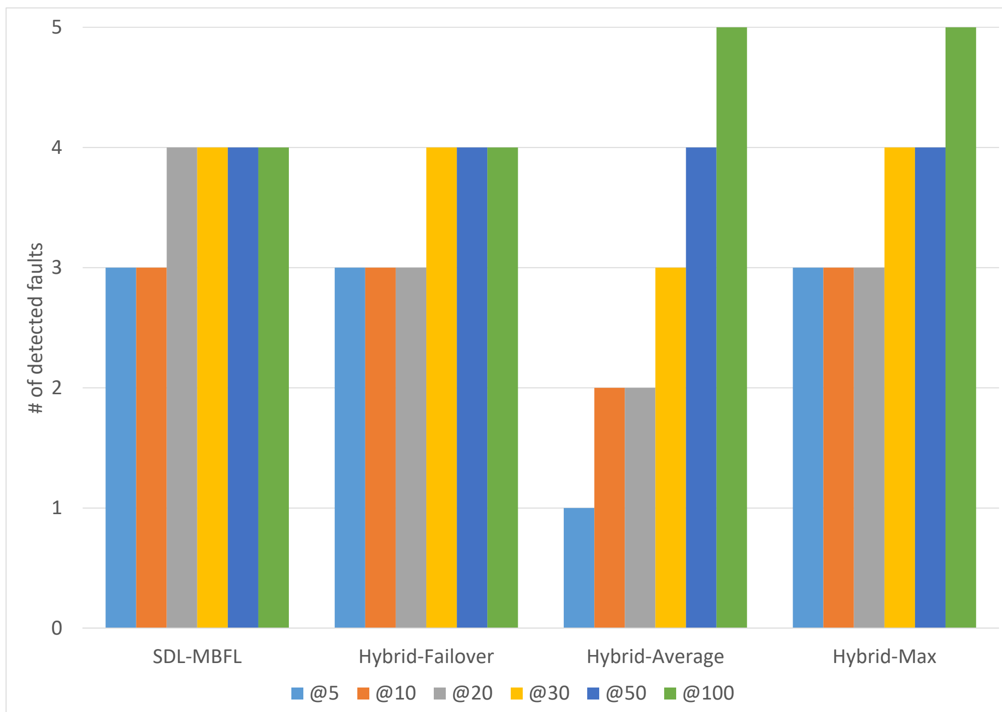

Figure 5.7: $E_{inspect}@n$ for each hybrid fault localization technique

Table 5.5: Results of fault localization (rank and average EXAM by $E_{inspect}$)

Fault ID	SDL-MBFL (Tarantula)	SDL-MBFL (Ochiai)	SDL-MBFL (Op2)	SDL-MBFL (DStar)	SBFL	MBFL	MBFL-w/o-SDL	Hybrid-Failover	Hybrid-Average	Hybrid-Max
1	13.0	3.3	10.2	3.3	9.0	19.0	634.2	23.2	7.0	25.2
2	1.3	634.0	663.0	663.0	181.0	1.9	634.2	2.2	48.0	2.4
3	1.3	3.3	10.2	3.3	102.0	1.9	284.0	2.2	29.0	2.4
4	643.5	645.5	658.5	658.5	604.0	788.5	301.0	1535.5	904.0	866.0
5	1645.0	1645.0	2230.0	1645.0	108.0	262.0	81.0	2562.2	620.0	312.0
6	515.5	254.0	183.5	237.8	70.0	617.7	634.2	1248.8	302.5	283.0
7	515.5	254.0	183.5	237.8	13.5	363.0	133.0	1248.8	80.5	87.5
8	1.3	447.0	428.0	428.0	2.2	1.9	4.0	2.2	1.8	2.4
9	1645.0	1645.0	2230.0	1645.0	1537.0	1991.5	634.2	1509.0	2281.5	2274.0
EXAM	0.101	0.112	0.133	0.112	0.053	0.082	0.068	0.164	0.086	0.078

5.5 Discussion

5.5.1 Practical cost-effectiveness

The execution time of a fault localization technique is an important consideration in its practical use. In this study, by limiting the mutation operator to SDL only, the execution time was reduced by approximately 20% while maintaining the same fault localization performance as that of a general MBFL. Meanwhile, SBFL differs from MBFL in that it can compute suspiciousness values in only one test run; therefore, the run is completed in a few seconds, and its fault localization performance is inferior to SDL-MBFL in $E_{inspect}$@100, which indicates the number of faults in the 100th position, but it outperforms the other methods in EXAM score, which indicates the average. In this section, we discuss how the fault localization technique should be utilized in practical applications by considering these characteristics.

Because mutation analysis is generally time consuming, the effect of MBFL on the execution cost has been discussed in past studies [211, 169, 126]. According to [118], less than 9% of practitioners are willing wait more than 1 hour to obtain the results of fault localization, and conversely, more than 90% would be satisfied with less 1 minute of execution time.

One of the ideas to alleviate the problem of execution time in practical use, is to incorporate it into automated testing as part of a continuous integration (CI) loop and to execute fault localization by using test failures as a trigger. In this work flow, fault localization is processed in the background without interfering with the developer's work, and the user can obtain the result of fault localization by notification from the CI when fault localization is completed. As reported in [118], such an approach received positive comments from several practitioners. A similar approach has been proposed for automated program repair [196].

We also believe that techniques for reducing execution time, such as omitting mutants on statements that are only executed by passed test cases, as well as test case selection and mutant reduction, are necessary for further scalability. However, if the reduction of testing and mutants results in compromising the fault localization performance, it may be necessary to determine what is acceptable in practice. In the experiment in this chapter, not only a single statement, such as a function call, but also a multi-line statement, such as an `if` command, are subject to SDL mutation, so it takes more time to finish. The difficulty in reducing these mutants is due to the large impact on the defect localization performance; as discussed in Subsection 5.5.3, searching for mutation operators that can replace them and reduce the number of mutants even more is future work.

5.5.2 Characteristics of the faults

In this thesis, we evaluate localization of nine faults at one time. It is known that SBFL for multiple faults is more difficult than SBFL for a single location [50, 64]. The main reason is that each fault affects each of the others and complicates the relationship between the test result and the fault. For example, if one fault causes an error in the data stored in the memory, another fault may overwrite and conceal the error in the data during propagation to the output. In addition, as described in Section 5.3.3, the faults under evaluation in this chapter imply incompatibilities before and after system re-engineering. Because there are some incompatibilities that do not require correction, and because the developers of the subject system do not disclose how to fix them, it is not possible to consider a scenario in which the faults are fixed one-by-one. For such scenarios, to debug

multiple faults in parallel, a clustering method [103] and an integer programming method [93] have been proposed.

In the paper that proposed the MUSE [141] MBFL technique, it is experimentally shown that the localization succeeded even in the case where multiple faults exist. The reason for the success is that the effect of mutation location on the output can be examined independently for each statement.

Fault IDs 2 and 3 are not ranked high in SBFL but are ranked high in MBFL. Including the fault ID1, they exist in the same function, and there is a fault ID3 in the part after the fault ID2 and a fault ID1 after part of the fault ID3. Listing5.1 and Listing5.2 present the source code parts of fault ID2 and fault ID3, respectively, after migration, with minor changes (such as variable names) that do not interfere with understanding.

Listing 5.1 Fault ID2

```
442 switch (isPart) {
443 case 0:
444     /* no partial size check */
445     break;
446 case 1:
447     if (! ((Partial_size == 0) ||
448         (Partial_size >= PART_SIZE_MIN &&
449          Partial_size <= PART_SIZE_MAX))) {
450         return -1;
451     }
452     break;
```

Listing 5.2 Fault ID3

```
507 if (! (port >= 0 &&
508        port <= 65535)) {
509     return -1;
510 }
```

With regard to fault ID2, the pre-migration program processed lines 447 to 451 regardless of the value of `isPart`; thus, it was incompatible to not process lines 447 to 451 when `isPart` is 0 in the post-migration program. By considering the test, we determined passed or failed by the value of `Partial_size` when `isPart` was 0, but localization was difficult in the SBFL because the passed test cases were more numerous than the failed test cases. In SDL-MBFL, deleting the statement `break` by statement deletion mutation results in a fall through, and when `isPart` is 0, the behavior is the same as when `isPart` is 1. In other words, this is the same behavior as before the migration, so SDL-MBFL can be appropriately localized.

Fault ID3 was improved to be checked by the post-migration program, but it was incompatible because the pre-migration program did not check the value of `port`. Faults in the conditional branches are less likely to be localized in SBFL because both passed and failed test cases run through them. In SDL-MBFL, the mutation that deletes the entire `if` statement from line 507 makes the behavior equivalent to that of the pre-migration program, so the localization is successful.

However, for fault IDs 5, 6, and 7, the fault localization performance of SDL-MBFL is significantly inferior to that of SBFL. Among them, ID7's $E_{inspect}$ has

high performance with SBFL at 13.5, while SDL-MBFL's performance is inferior at 515.5. This can be regarded as a fault that most expresses the characteristics of their strengths and weaknesses. The source code of fault ID7 is shown in Listing5.3, with minor changes that do not interfere with understanding. In the pre-migration program, the error code when `smtp_data()` fails was returned as-is regardless of the result of `smtp_auth()`, whereas in the post-migration program, if `smtp_auth()` fails and `smtp_data()` also fails, the error code in `smtp_auth()` is returned, thus creating an incompatibility. SDL-MBFL measures the effect of the test result when the statement at line 184 is deleted, but because the value of `error` is already set to `auth_error` at line 180, the output does not change even when the statement is deleted, which makes it difficult to localize.

Listing 5.3 Fault ID7

```
179 if ((rc = smtp_auth()) < 0) {
180     auth_error = error = rc;
181 }
182 if ((rc = smtp_data()) < 0) {
183     if(auth_error != 0){ /* authentication error */
184         error = auth_error;
185     }else{
186         error = rc;
187     }
188     return error;
189 }
```

In this experiment, we implemented SDL-MBFL based on Metallaxis, which is one of the most prominent methods, partly because it is not intuitive to change a failed test case to a passed test case by statement deletion mutation alone, and we believe that MUSE, the other most prominent method, is not suitable for this purpose. However, the failure IDs 2 and 3 may also be ranked by the MUSE-based SDL-MBFL because the failed test cases become passed ones.

5.5.3 How to choose a mutation operator

The observation of the localization of fault ID7 shows that it is necessary to devise the selection of mutation operators. Rather than deleting for all types of statements, as in the SSDL in [3], we suppose that deleting one-line statements, such as fault IDs 2 and 3, and deleting the condition of an `if` statement, such as fault ID7 (i.e., always true), would be more effective. This idea is similar to the combination of the Void Method Call Mutator and the Remove Conditionals Mutator in Pit [44]. SSDL also deletes for compound statements, such as `if` and `for` statements, but such deletes have a large scope of influence and may be difficult to localize. Furthermore, such deletions can be disadvantageous in localization because their effect on the conditions in the `if` statement cannot be measured. However, there are not yet many observed cases, so further investigation of generality is needed.

5.6 Summary

In this chapter, a mutation-based fault localization using a statement deletion mutation operator was proposed and evaluated using nine faults in real industrial software. As a result of this evaluation, SDL-MBFL had the highest fault

localization performance in Tarantula's formula, and the number of detected faults with higher rankings was higher than that of SBFL and MBFL without SDL. In the hybrid method with SBFL, Hybrid-Max, a method that selects the maximum value of the suspiciousness, performed better overall. Although the execution time was reduced by 20.3 % compared to MBFL, it is still a long time, so we concluded that it is necessary to incorporate it into continuous integration to make it practical.

Future works include quantitative comparisons with MBFLs using other mutation operators and execution time reductions through combinations with other mutation reduction methods and test case reduction methods. We also need to work on further evaluation of software within enterprises.

Chapter 6

Error-Oriented Mutant Reduction and Mutant Weighting for Reliable Mutation Analysis

6.1 Overview

Because the naïve mutation analysis requires running a test suite for each mutant, the execution time is determined by the product of the test execution time and the number of mutants. It is important to know how to reduce the number of mutants to scale the mutation analysis.An important aspect of the approach to reducing the number of mutants is that the measurement of fault detectability after mutant reduction is almost the same as before mutant reduction, that is, the mutation score for fault detectability is almost unchanged. For example, if 6 out of 10 mutants were able to kill in the test before the mutant reduction (i.e., the mutation score is 60%) but 5 out of 5 mutants are able to kill in the test after the mutant reduction (i.e., the mutation score is 100%), the mutation score will be far from the original 60%. This means that mutant reductions will make it impossible to measure fault detectability accurately. A person who is informed of an inaccurate fault detection result may mistakenly perceive that the testing has been done sufficiently well even though the testing has not sufficiently squashed the potential faults in the program, or vice versa, potential faults may be squashed by testing but misidentified as poorly tested.

In the mutant reduction model of Ammann et al. [8], mutants killed by the same test case combination are considered redundant because they do not contribute to the measurement of fault detectability, leaving one behind and deleting it. However, as a unit of measurement of mutant redundancy, the classification of a test case-by-test case basis might be coarsely grained. When considering that one test method is equivalent to one test case in unit testing, there are many cases in which various properties are tested in one test case by calling multiple test target methods in one test method and checking the results by multiple assertions. It would be more natural not to view the set of mutants killed in such a test case as homogeneous. In addition, test code refactoring [61], which was proposed in the early 2000s, has been recognized in recent years, and it is recommended to refactor test code to avoid anti-patterns that contain multiple assertions in one test case, which is called assertion roulette, because it has a bad influence on program understanding at the time of maintenance [20, 21]. This is illustrated by the experimental results that 45.7% of the assertions in the test code do not contribute to the detection of faults, indicating the need to organize the assertions in the test code [213]. Against this background, it is important to keep mutants that can be used to check whether the modification of statements in

the test method, such as the deletion of assertions, does not result in overlooking future faults.

In this chapter, we present a method for determining mutant redundancy by the type of errors caused by mutation. We then introduce a model that reduces mutants with overlapping types of errors by treating them as redundant. In addition, we propose a way to calculate the impact of reduced mutants and weight the remaining mutants. Consequently, we were able to reduce the discrepancy between mutation score after mutant reduction and that of the original mutant set, which means that our weighted mutant reduction technique can measure the mutation score more accurately.

These techniques allow us to measure the same fault detectability in the reduced mutant set as in the original mutant set, even if the statements in the test code are modified or deleted by refactoring or other means. We also tried to reduce the number of mutants for 53 projects in OSS and evaluated the number of mutants and execution time. Furthermore, for three of those projects, we evaluated how well the proposed model and existing model could measure the correct fault detectability close to that before the mutant reduction when the assertions in the test code were removed at a certain rate to reduce the fault detectability.

The structure of this chapter is as follows. In Section 6.2, we introduce the preliminary background materials. In Section 6.3, we give an example of our motivation. In Section 6.4, we explain the proposed method. In Section 6.5, we present the evaluation method and results, and in Section 6.6, we discuss the results.

6.2 Preliminary

The test case-based mutant reduction model by Ammann et al. [8] defines the minimal test set that can maintain a mutation score as follows.

Definition 6.1. Let M be a finite set of mutants and T be a finite test set on some program P. A test set $\hat{T}_M$ is minimal if and only if for any test $tc_i \in \hat{T}_M$, $\hat{T}_M - \{tc_i\}$ does not maintain the mutation score with respect to original M and T. Let $\bar{T}_M = \{\hat{T}_1, \hat{T}_2, ...\}$ denote the set of all possible minimal test set with respect to M.

This means that tests that do not contribute to improving the mutation score are regarded as redundant.

Table6.1 shows an example with five test and four mutants: $T = \{tc_1, tc_2, tc_3, tc_4\}$ and $M = \{m_1, m_2, m_3, m_4\}$. The "t"s in the table mean that the corresponding test can kill the corresponding mutant, i.e., tc_1 can kill three mutants, m_1, m_2, and m_4. The test set T is capable of killing all mutants in M, and there are three minimal test sets $\bar{T}_M = \{\{tc_4\}, \{tc_1, tc_2\}, \{tc_1, tc_3\}\}$, that can kill all mutants, but if any of the included tests are lacking, not all mutants can be killed.

Table 6.1: Example of mutants and test

	m_1	m_2	m_3	m_4
tc_1	t	t		t
tc_2		t	t	
tc_3	t		t	
tc_4	t	t	t	t
tc_5	t	t		

Using the concept of a minimal test set, we define a redundant mutant as follows.

Definition 6.2. Let $M_j = M - \{m_j\}$ for some mutant $m_j \in M$. We say that m_j is redundant with respect to mutant set M and test set T if and only if $\bar{T}_M = \bar{T}_{M_j}$.

This means that mutants that do not affect the minimization of the test set will be considered redundant. In other words, if a mutant is removed and the corresponding test is no longer minimal, we can consider that mutant as non-redundant.

Here is a running example using Table 6.1. Computing T_M first for full mutant set M, and then T_{M_i} for M excluding each mutant m_i, we get the following.

$$
\begin{aligned}
\bar{T}_M &= \{\{tc_4\}, \{tc_1, tc_2\}, \{tc_1, tc_3\}\} \\
\bar{T}_{M_1} &= \{\{tc_4\}, \{tc_1, tc_2\}, \{tc_1, tc_3\}\} \\
\bar{T}_{M_2} &= \{\{tc_4\}, \{tc_1, tc_2\}, \{tc_1, tc_3\}\} \\
\bar{T}_{M_3} &= \{\{tc_1\}, \{tc_4\}\} \\
\bar{T}_{M_4} &= \{\{tc_4\}, \{tc_1, tc_2\}, \{tc_1, tc_3\}, \{tc_2, tc_5\}, \{tc_3, tc_5\}\}
\end{aligned}
$$

Note that $\bar{T}_M$, $\bar{T}_{M_1}$, and $\bar{T}_{M_2}$ are identical, that means both m_1 and m_2 are redundant with respect to M.

We further define a minimal mutant set as follows.

Definition 6.3. A mutant set M is minimal if it contains no redundant mutants.

In Table 6.1, the minimal mutant set is $\{m_3, m_4\}$.

6.3 Motivating Example

To introduce our mutant reduction model, we present an example of a problem with the existing method and demonstrate the need to invent a new method. We now consider testing the isEmpty method of Apache Commons Lang.

In the test target method isEmpty() shown in the following listing 6.1, there are four mutants in total: two mutation operators that invert the comparison operator and two mutation operators that replace the left and right sides of the logical sum with false. m_1 in listing 6.2 and m_2 in listing 6.3 become mutants on the left side of the logical sum, and similarly, m_3 in listing 6.4 and m_4 in listing 6.5 become mutants on the right side of the logical sum.

Listing 6.1 Example of original source code (StringUtils.java)

```java
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || cs.length() == 0;
3 }
```

Listing 6.2 Example of mutant m_1

```java
1 public static boolean isEmpty(CharSequence cs) {
2   return cs != null || cs.length() == 0;
3 }
```

Listing 6.3 Example of mutant m_2

```
1 public static boolean isEmpty(CharSequence cs) {
2   return false || cs.length() == 0;
3 }
```

Listing 6.4 Example of mutant m_3

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || cs.length() != 0;
3 }
```

Then, all mutants can be killed by the test method given by the listing 6.6. However, the statements that cause the test failure are lines 2-4 in the test code, and if we remove all statement other than line 2, we cannot kill all the mutants.

Since one test method corresponds to one test case, in case of test case-based mutant reduction model, a single test method in listing 6.6 can kill all mutants, so the single test method becomes the minimum test set. That is, if all four mutants can be killed by the single test method, all but one mutant of the four will be considered redundant and reduced.

Now, consider the mutation score for a test method (listing 6.7) that weakens fault detectability by leaving only the second line of the test method statement in listing 6.6.

The test method with weakened fault detectability can only kill two of the four mutants: m_1 and m_2, which means that the mutation score is reduced to 50%. Here, if the test case-based reduction model leaves only m_1 out of the four mutants reduced, this mutant can be killed even by a test method with weakened fault detectability, so the mutation score remains at 100%, a difference from the true mutation score before the mutation reduction.

Thus, it may not be possible to accurately measure the fault detectability without considering the effect of each statement in the test method instead of each test case. In particular, the mutant reduction on a per-test-case basis may not be able to detect a decrease in fault detectability, especially if test code refactoring causes modifications on a statement-wise basis in the test code.

6.4 Proposed Method

As shown in Section 6.3, the purpose of our method is to find the minimal amount of mutants that can detect a change in the mutation score, that is, the fault detectability due to modification when a statement-wise modification occurs in the test method.

Errors caused by mutations occur during the execution of the test target method called in the test method or by assertions after the execution of the test target method. The stack trace in the error records the call hierarchy from the statement in the test method to the point where the error occurred. That is,

Listing 6.5 Example of mutant m_4

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || false;
3 }
```

Listing 6.6 Example of test code (StringUtilsTest.java)

```
1 @Test public void testIsEmpty() {
2   assertTrue(StringUtils.isEmpty(null));
3   assertTrue(StringUtils.isEmpty(""));
4   assertFalse(StringUtils.isEmpty(" "));
5   assertFalse(StringUtils.isEmpty("foo"));
6   assertFalse(StringUtils.isEmpty("  foo  "));
7 }
```

Listing 6.7 Example of test code with weakened fault detectability

```
1 @Test public void testIsEmpty() {
2   assertTrue(StringUtils.isEmpty(null));
3 }
```

we can tell which mutants are affecting the statements in the test method by observing the errors caused by the mutants.

The method proposed in this chapter minimizes the number of mutants by defining the redundancy for errors that occur in mutation analysis and eliminating the redundant mutants. Furthermore, we aim to reduce the discrepancy from the original mutation score by weighting mutants that are close in nature to those that are reduced.

Note that all mutants treated by our method are killed by the test set T. Mutants that are not killed by the test set T are out of the reduction because they do not have information, such as "which test killed them'' or "what errors occurred,'' and it is not possible to examine the relationship between mutants and tests or errors. In terms of knowing the trend of the reduction model, it is critical to have a sufficient number of mutants killed in a statistical manner, and we prepare a test set T that can kill such a sufficient number of mutants.

6.4.1 Definitions

For the new method, we define a "redundant mutant" that differs from the definition given in Section 6.2. First, for this purpose, we define an error.

Definition 6.4. An error consists of a combination of exception type ex and stack trace st.

The type of exception gives information to determine what kind of event has occurred when an error occurs, for example, NullPointerException in Java. The stack trace is a record of the method call hierarchy from the place where the error occurred.

We define an "identical error" as one whose exception type and stack trace are identical as follows.

Definition 6.5. Let $err_i = err_j$ if the type of exception ex_i of error err_i and stack trace st_i and the type of exception ex_j of error err_j and stack trace st_j are equivalent. Note that the stack traces are equivalent if the call method name, file name, and its line number in each layer of the stack trace are equivalent.

We further define "error distinguishable'' in relation to the mutant of the error cause as follows.

Definition 6.6. Let $E_{M,T}$ be the set of errors that occur during executions of test T on mutant set M. We say that err_i and err_j be distinguishable if error $err_i \in E_{M,T}$ and error $err_j \in E_{M,T}$ is not equivalent and if mutant set $M' \subseteq M$ raising err_i and mutant set $M'' \subseteq M$ raising err_j are not equivalent. If any two different errors err_i and err_j in error set $\hat{E} \subseteq E_{M,T}$ are distinguishable, the error set $\hat{E}$ is called distinguishable.

We define a function to delete redundant errors that are not distinguishable.

Definition 6.7. If $\hat{E} \cup \{err\}$ is not distinguishable for some distinguishable error set $\hat{E} \subseteq E_{M,T}$ and another error $err \in E_{M,T} \setminus \hat{E}$ not included in the set, then err is called a redundant error. Let $\mathcal{D}(E_{M,T})$ be a set of errors without redundant errors from error set $E_{M,T}$.

Suppose we remove mutants while keeping the redundancy-free error set distinguishable. If the error set is distinguishable, it is possible to determine which mutants affect the statements in the test method.

Based on the concept of Definition 6.2, we define redundant mutants for error set $E_{M,T}$ as follows.

Definition 6.8. If $\mathcal{D}(E_{M,T})$, where redundant errors are removed from the error set $E_{M,T}$ of the mutant set M, and $\mathcal{D}(E_{M_j,T})$, where redundant errors are removed from the error set $E_{M_j,T}$ of the mutant set M_j, are equal, i.e., $\mathcal{D}(E_{M,T}) = \mathcal{D}(E_{M_j,T})$, then m_j is called redundant in the error set $E_{M,T}$.

In Definition 6.3 for the minimal mutant set, the set of mutants that contains no redundant mutants is regarded as the minimal, so the model of the minimal mutant set for the error set $E_{M,T}$ can be obtained by deleting the redundant mutants defined in Definition 6.8.

6.4.2 Mutant Set Minimization Algorithm

In this chapter, we optimize the execution time of the entire mutation analysis by selecting the mutants with short execution times when obtaining the minimal mutant set. Concretely, we first execute each mutant m and measure the test execution time $time_{m,T}$ for each mutant $m \in M$ and obtain the raised error set $E_{\{m\},T}$. Then, by solving the following optimization problem, we obtain the mutant set $\hat{M}$ with the shortest execution time that covers all errors.

- Input: set of killed mutants M_{killed}, test set T, total error set $E_{M_{killed},T}$
- Output: mutant set $\hat{M}$
- Constraints: $e_i \neq e_j (\forall e_i, e_j \in E_{\hat{M},T})$
- Objective function: $\sum_{m \in \hat{M}} time_{m,T}$
- Goal: minimization

The combination of errors satisfying the constraint is $\frac{|E_{\hat{M},T}| \cdot (|E_{\hat{M},T}|-1)}{2}$, and the combination of the mutant set $\hat{M}$ in the output is $2^{|\hat{M}|}$. Finding the optimal solution requires exponential order computation, so we find the minimal set of mutants by using a greedy method that selects the one with the shortest execution time among the mutants that generate the least errors. The details are given in Algorithm 1.

Algorithm 1 Mutant Minimization by Greedy Algorithm

Input: M: Mutant set, T: Test set, $E_{M,T}$: Error set
Output: $\hat{M}$: Minimized mutant set
  $\hat{M} \leftarrow \phi$
  $E_{cover} \leftarrow \phi$ ▷ A set of selected errors
  while $E_{cover} \neq E_{M,T}$ **do**
    $M_{minerr} \leftarrow \arg\min_{m \in M \setminus \hat{M}} |E_{\{m\},T} \setminus E_{cover}|$ ▷ The set of mutants with the fewest unselected errors
    $m' \leftarrow \arg\min_{m \in M_{minerr}} time_{m,T}$ ▷ Select the mutant with the shortest execution time
    $M \leftarrow M \setminus \{m'\}$
    if $E_{m',T} \setminus E_{cover} \neq \phi$ **then**
      $\hat{M} \leftarrow \hat{M} \cup \{m'\}$
      $E_{cover} \leftarrow E_{cover} \cup E_{\{m'\},T}$
    end if
  end while
  return $\hat{M}$

6.4.3 Mutant Weighting

Because the mutant set minimized by the method presented in the previous section will be smaller than the original mutant set, the existing calculation method will result in erroneous mutation scores due to differences in population. Therefore, we propose a method to reduce the difference in mutation score. This method weights the impact of the removed mutants on the remaining mutants to simulate the scoring in the original mutant set, thereby calculating a more accurate mutation score. The weighting method is shown in Algorithm 2.

Algorithm 2 Mutant Weighting

Input: M: Mutant set, $\hat{M}$: Minimized mutant set
Output: $w_{m_1}, ..., w_{m_n}$: A set of mutant weights
  $w_{m_1}, ..., w_{m_n} \leftarrow \{1, ..., 1\}$ ▷ Initialization of weights
  for $m_{removed} \in M \setminus \hat{M}$ **do** ▷ $m_{removed}$: A removed mutant
    $m_{nearest} \leftarrow \arg\min_{\bar{m} \in \hat{M}} |E_{\{m_{removed}\},T} \oplus E_{\{\bar{m}\},T}|$ ▷ Mutants with the smallest symmetric difference in the error set
    $w_{m_{nearest}} \leftarrow w_{m_{nearest}} + w_{m_{removed}}$
    $w_{m_{removed}} \leftarrow 0$
  end for
  return $w_{m_1}, ..., w_{m_n}$

The algorithm is designed to give more weight to the remaining mutants that have a closer impact on the statements in the test method to the removed mutants. For this purpose, we select a mutant that make symmetric difference between the error set of the remaining mutant and the error set of the removed mutant minimal, that is, the error set of the remaining mutant is similar to the error set of the removed mutant. Because the total output weights are the same as the number of mutants in the original mutant set, we can compute a more accurate mutation score by reducing the difference due to differences in the population. Let $\hat{M}_{killed}$ be the set of mutants killed by test set T' out of

the minimized mutant set $\hat{M}$, where the mutation score can be expressed by the following equation:

$$MutationScore(\hat{M}, T') = \frac{\sum_{m \in \hat{M}_{killed}} w_m}{\sum_{m \in \hat{M}} w_m} \tag{6.1}$$

6.4.4 Example of Mutant Set Minimization

Here, we minimize for a mutant, as shown in Table 6.2. Let all mutants $m_1, ..., m_4$ be killed, and let the errors that occur in those mutants be $err_1, ..., err_4$. The relationship between a mutant and the error caused by it is represented by t. Moreover, the bottom row of Table 6.2 shows the execution times of all tests for each mutant.

Table 6.2: Example of mutants and errors

	m_1	m_2	m_3	m_4
err_1	t	t		
err_2	t		t	t
err_3	t		t	
err_4	t		t	
execution time	1	3	2	2

To obtain a distinguishable set of errors, we first explain the removal of redundant errors. Because err_3 and err_4 are caused by the same mutant, they can be seen as indistinguishable. Therefore, either err_3 or err_4 can be removed. In this case, err_4 is removed as redundant.

The next step is to select a mutant. In the greedy method shown in Algorithm 1, we choose the mutants with the fewest errors to cause, so here, m_2 and m_4 are the candidates to be left. Comparing the execution times of m_2 and m_4, m_4 is smaller, so m_4 is chosen first. Next, m_2 is chosen in order of the number of errors to be made, followed by m_3. At this point, because all errors are covered, the algorithm is finished and outputs the selected m_2, m_3, m_4.

From the perspective of optimization, it can be seen that the solution output by this algorithm is non-optimal because the execution time is less for deleting m_2 than for deleting m_1.

6.4.5 Example of Mutant Weighting

We illustrate the weighting of mutants using the example used in the previous section. We first give each mutant a weight of $w_{m_1}, w_{m_2}, w_{m_3}, w_{m_4} = 1, 1, 1, 1$ as the initial state. We then add weights to the mutants whose error set has the smallest symmetric difference with the error set of the removed mutant. Considering the removed mutant, m_1, it has the error set $\{err_1, err_2, err_3, err_4\}$, which is the nearest to the error set $\{err_2, err_3, err_4\}$ of the remaining mutant m_3, i.e., the smallest symmetric difference, so m_3 is selected as the weighting target. Actually, the size of the symmetry difference between E_{m_1} and E_{m_3} is $|E_{m_1} \oplus E_{m_3}| = |\{err_1\}| = 1$, which is smaller than the symmetry difference with the error sets of other remaining mutants, as $|E_{m_1} \oplus E_{m_2}| = |\{err_2, err_3, err_4\}| = 3$ and $|E_{m_1} \oplus E_{m_4}| = |\{err_1, err_3, err_4\}| = 3$. Therefore, by adding the weight of m_1 to the weight of m_3, we get $w_{m_1}, w_{m_2}, w_{m_3}, w_{m_4} = 0, 1, 2, 1$. If the test suite is updated and as a result mutants m_1 and m_3 are not killed, but m_1 is removed and cannot be found alive, the mutation score using weighting is $\frac{w_{m_2}+w_{m_4}}{w_{m_2}+w_{m_3}+w_{m_4}} = 0.5$,

which can be calculated without m_1. This can be rephrased as predicting whether m_1 is killed through m_3, which is nearest to m_1. If weighting was not used, the mutation score would be $\frac{|(m_2,m_4)|}{|(m_2,m_3,m_4)|} = 0.66...$, which would result in a difference from the previous mutation score before minimization $\frac{|(m_2,m_4)|}{|(m_1,m_2,m_3,m_4)|} = 0.5$. We can see that the difference can be reduced by using weighting.

6.5 Evaluation

6.5.1 Research Questions

To check the effectiveness of the proposed method, we investigated the following four research questions.

- RQ1: How much statement-wise modification actually occurs in the test method?
- RQ2: How much is execution time reduced?
- RQ3: How much less of a difference in mutation score can we achieve?
- RQ4: Which mutation operators are reduced most?

The purpose of each research question is explained below. RQ1 aims to show the importance of the problem solved by the proposed method by showing how real the problem of statement-wise modification in the test method is. The purpose of RQ2 is to show the practicality of the proposed method by investigating the performance difference between the proposed method and the existing method, even though the proposed method may not reduce many mutants compared to the existing method. The purpose of RQ3 is to show the advantage of the proposed method by investigating the difference in mutant score between the proposed method and existing method because the proposed method is expected to be able to measure the mutant score close to the one without mutant reduction. RQ4 aims to determine whether mutation operators can be used as a guide for making reductions on an unknown set of mutants by examining the tendency of mutation operators in selecting the best mutants.

6.5.2 Evaluation Method

To realize the proposed method presented in Section 6.4, the following features are implemented in the mutation analysis tool PIT [44].

- Record exceptions and stack traces associated with mutants when tests on mutants fail.
- Continue to run other tests even if the test fails on mutants.
- Record each mutant's execution time.
- Replay the recorded mutant set.

The mutation operators used in the PIT are listed in Table 6.3.

Using the exceptions, stack traces, and execution times for each mutant recorded by these features, we find the minimal mutant set from the set of killed mutants.

As a comparison, we also obtained the test case-based minimal mutant sets and measured the execution time and mutation score for each minimal mutant set

Table 6.3: Mutation operators used for the evaluation

Type	Name	Acronym
Default	Conditionals Boundary Mutator	CBM
	Increments Mutator	IM
	Invert Negatives Mutator	INM
	Math Mutator	MM
	Negate Conditionals Mutator	NCM
	Return Values Mutator	RVM
	Void Method Calls Mutator	VMCM
Non-Default	Constructor Calls Mutator	CCM
	Inline Constant Mutator	ICM
	Member Variable Mutator	MVM
	Non Void Method Calls Mutator	NVMCM
	Remove Conditionals Mutator	RCM
	Remove Increments Mutator	RIM
	Switch Mutator	SM
	Argument Propagation Mutator	APM
	Naked Receiver Mutator	NRM
	Remove Switch Mutator	RSM

from mutation analysis. The mutation score is measured by dropping a specified percentage of assertions in the test code to confirm whether the mutation score of the minimal mutant set is equal to that of the original mutant set even if the fault detectability of the test code is reduced. To avoid side-effects, the method call that is an argument in the assertion is not removed from the assertion so that the test code other than the assertion behaves the same as the original test code.

6.5.3 Subject of Evaluation

To evaluate the proposed method, we extract the subject code using the data of TravisTorrent [22], which is collected from the build data of Travis CI, as follows.

1. From the TravisTorrent data, select code from the master branch of a repository with Java language, Maven build tool, successful test code, and more than 1,000 lines of source code.
2. Leave the above code that terminates normally at the time of PIT execution.
3. Leave the above code that successfully terminates the execution of the recorded mutant.

The 53 repositories extracted in the above manner are listed in Table 6.4.

For these codes, the minimal mutant set is calculated. In addition, we measure the mutation scores during assertion reduction for three of these repositories: jsoup, zt-zip, and jInstagram. Table6.5 lists the numbers of test cases, numbers of assertions, and the average numbers of assertions per test case.

As a comparison, Gopinath et al.'s [86] test case-based mutant reduction method is used to obtain the minimal mutant set, and the mutation score during assertion reduction is measured in the same way. Then, we added weights to the mutants that have the smallest symmetric difference with respect to the test case set instead of the error set.

6.5.4 Evaluation Results

RQ1: How much statement-wise modification actually occurs in the test method?

It is important to show the significance of the problem solved by the proposed method in terms of how realistic the statement-wise modification in the test method, as shown in Section 6.3, is. To understand this perspective, we investigated whether the number of assertions changed or not and whether the number of test cases changed out of 11,080 commits, including modifications to the test code, in the 202 TravisTorrent projects collected in (1) of Section 6.5.3.

The results are shown in Figure 6.1. The left bar represents the number of commits that reduced the number of assertions, the middle bar represents the number of commits that increased the number of assertions, and the right bar represents the number of commits that did not change the number of assertions. Also, the red part of each bar is the number of commits that increased/reduced the number of test cases, and the blue part is the number of commits with no change in the number of test cases.

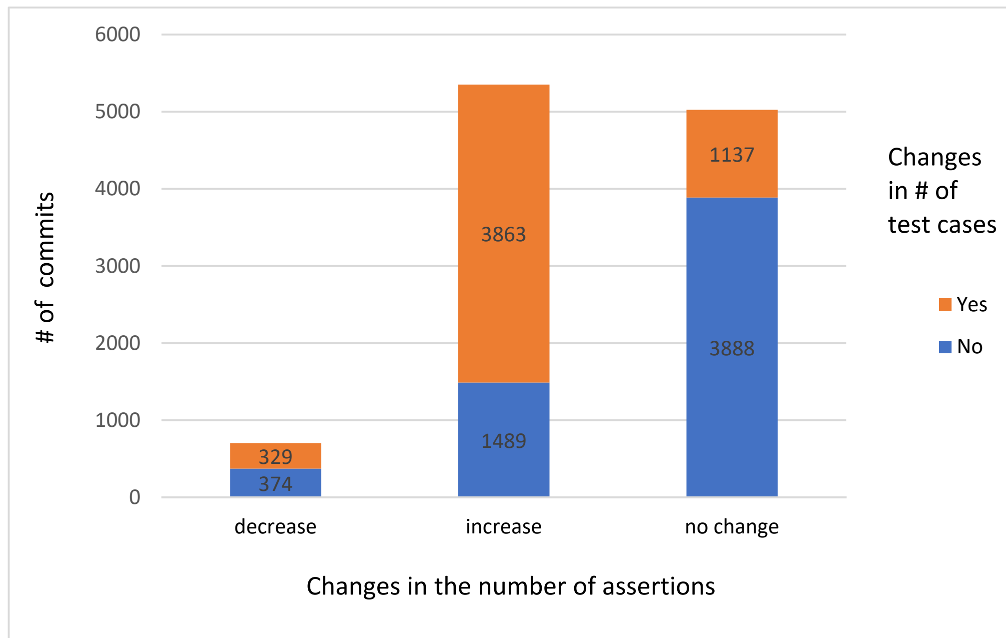

Figure 6.1: Ratio of modified assertions in commits of test code modification

The largest numbers of commits are those with an increasing number of assertions, followed by commits with no change in the number of assertions. The number of commits with decreasing assertions accounts for approximately 6% of the total, which is quite small compared to the other cases. The number of commits with statement-wise modifications on assertions in the test method, i.e., commits with no change in the number of test cases but with an increase or decrease in the number of assertions, is more than one quarter of the number of commits with an increase in the number of assertions, and more than half of the number of commits with a decrease in the number of assertions.

Furthermore, Figure 6.2 shows the distribution of the increase or decrease in the number of assertions in the test code modification commit, with and without the change in the number of test cases. For visibility, the upper limit of the y-axis is set to 1000, and the range of the x-axis is set to -50 to 50.

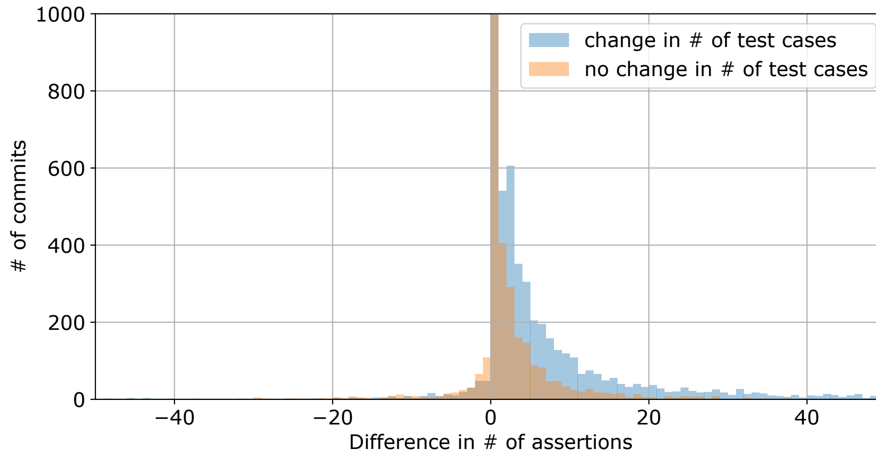

Figure 6.2: Distribution of increased/decreased assertions in commits of test code modification

The side of the x-axis greater than 0 indicates an increase in the number of assertions, and the side less than 0 indicates a decrease in the number of assertions; the case of 0 indicates no change in the number of assertions. In this figure, it can be seen that there is a small number of commits that reduce the number of assertions but also that the amount of assertions that are reduced is not as large as the increase.

RQ2: How much is execution time reduced?

First, the total execution times for all projects of the mutation analysis for the proposed method (error-oriented optimization), the existing method (test-oriented optimization), and the non-optimized case are listed in Table 6.6. Figure 6.3 shows the distribution of execution time ratios for each project compared to those without optimization. The left side of Figure 6.3 shows the distribution of error-oriented optimization and the right side is the distribution of test-oriented optimization.

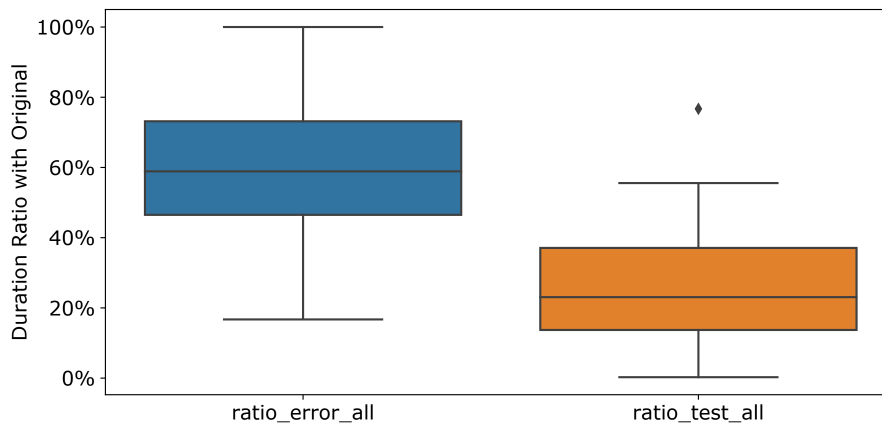

Figure 6.3: Distribution of execution time of mutation analysis

From these results, it can be seen that the error-oriented optimization of the proposed method reduces the execution time by approximately 60% compared to the non-optimized method, while the test-oriented optimization reduces the execution time by approximately 25%. This means that the proposed method does not reduce the execution time as much as the existing method of test-oriented optimization, but it allows mutation analysis to be executed in a shorter time than without any optimization.

RQ3: How much less of a difference in mutation score can we achieve?

Among the 53 repositories selected in Section 6.5.3, we measured the mutation scores of three repositories (jsoup, zt-zip, and jInstagram) when the statements in the test code were reduced by a certain percentage (80%), and we examined how little the difference from the original mutation score could be measured.

Table 6.7 shows the number of killed mutants, and Figure 6.4 shows the absolute value of the difference in mutation scores between the non-optimized and each reduction method.

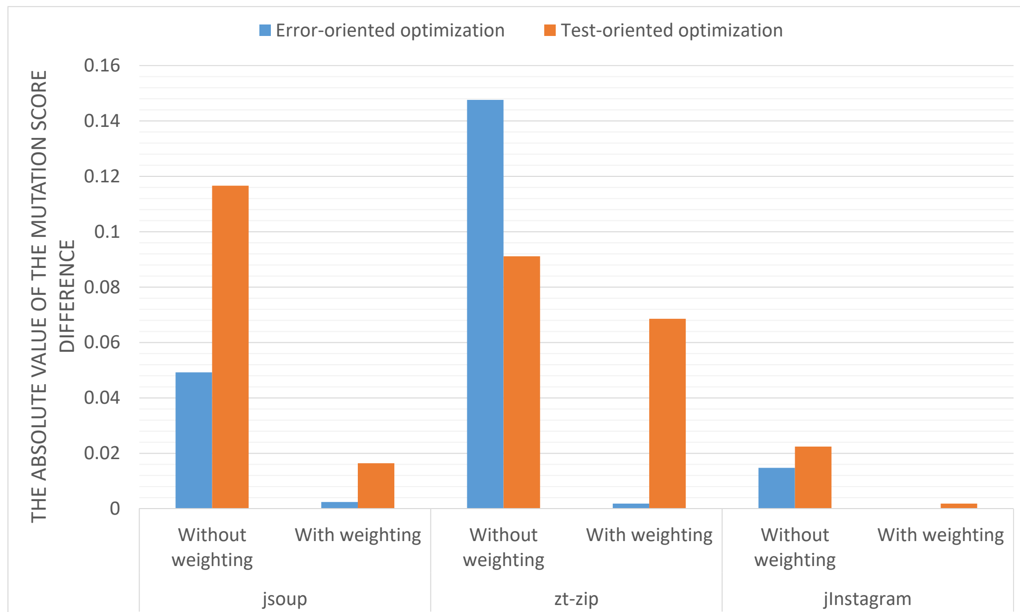

Figure 6.4: Absolute values of mutation score difference in reduction of assertions

As shown in Figure 6.4, the absolute value of the difference was greatly reduced in all three cases by weighting. Especially for the error-oriented optimization, the weighting resulted in the absolute value of the difference being less than 0.002 in both cases, which was better than the test-oriented optimization.

We also measured how the weighted mutation scores changed with varying rates of statement reduction in the test code of jsoup.

Table 6.8 lists the numbers of killed mutants, and Figure 6.5 shows the absolute value of the difference in mutation scores between the no-optimization and each reduction method.

Figure 6.5 shows that the error-oriented optimization makes less difference than the test-oriented optimization in the reduction rate of test code statements. This is especially evident when the test code statement reduction rate is 80%.

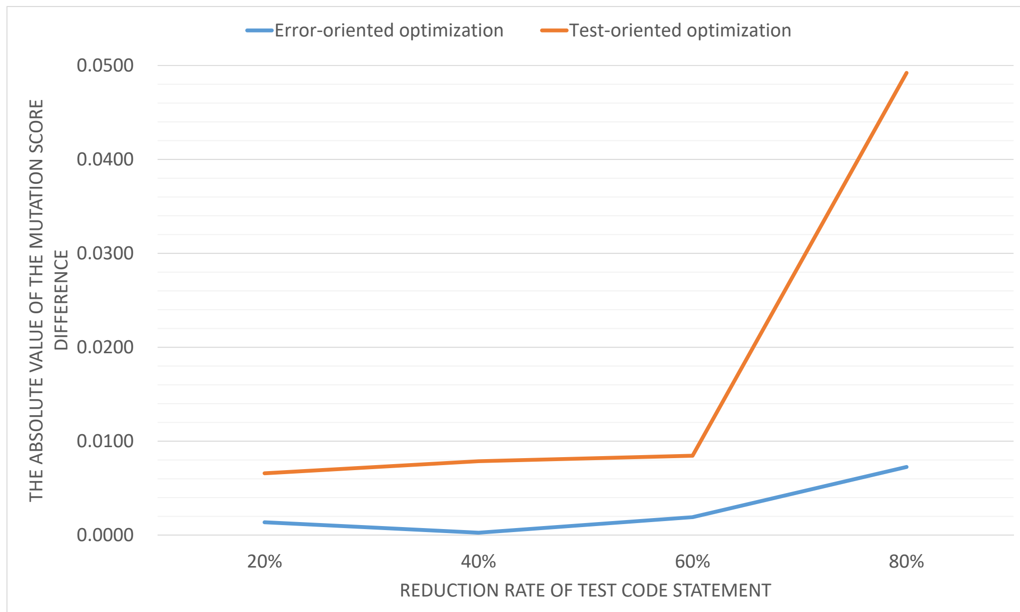

Figure 6.5: Absolute values of mutation score error in reduction of test code's statements in jsoup

RQ4: Which mutation operators are reduced more?

Knowing the trends of mutation operators reduced by the proposed method is important because it can guide the reduction for a mutant set that is unknown to be killed, such as when we perform mutation analysis on another new program. The results of our investigation into which mutation operators are reduced in large numbers by the proposed method are shown in Figure 6.6. The blue bars in Figure 6.6 represent the original mutant set, and the red bars represent the numbers of mutants in the reduced mutant set. Furthermore, the reduction rate (# of mutants targeted for reduction / total mutants) for each mutation operator is shown in Figure 6.7.

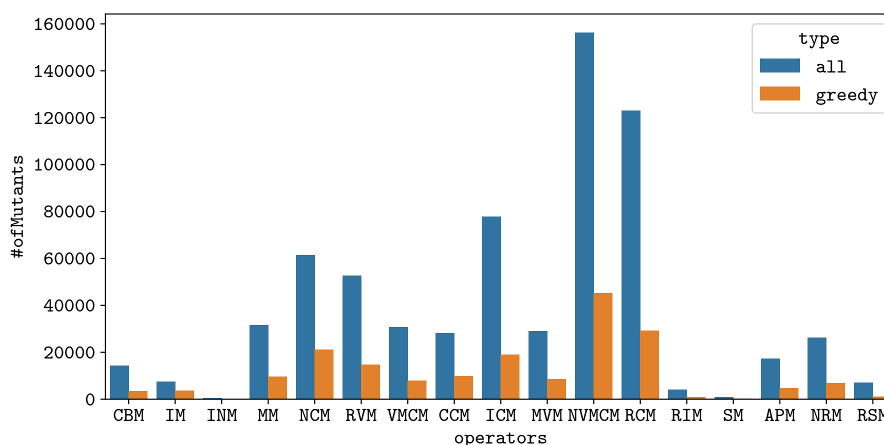

Figure 6.6: Number of mutants per mutation operator

It can be seen that the Non Void Method Calls Mutator has the highest number of mutants per mutation operator, unchanged before and after the reduction. Meanwhile, the mutant reduction rate by mutation operator was the lowest for

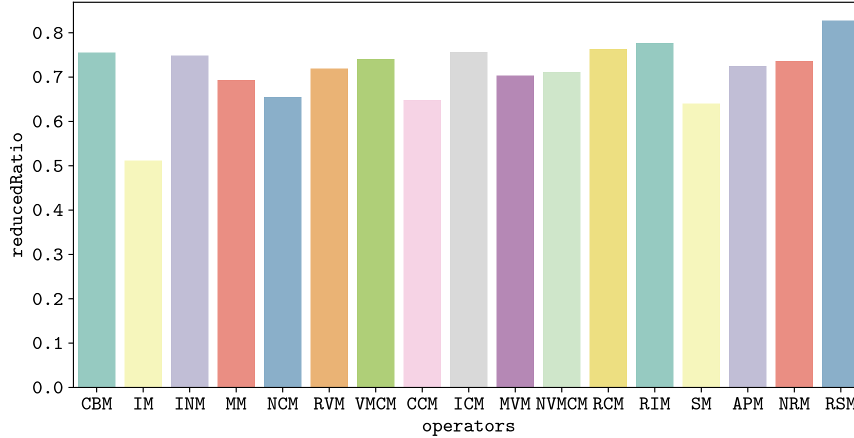

Figure 6.7: Reduction rate of mutants per mutation operator

the Increments Mutator and the highest for the Remove Switch Mutator, but overall, there was not much variance in the reduction rate, and none of the reduction rates could be said to be characteristic.

6.6 Discussion

6.6.1 Ratio of Assertion Fixes to Test Code Fix Commits

It is found that there is a small percentage of commits that reduce the number of assertions without increasing or decreasing the number of test cases. We sampled and observed several commits that reduced assertions, some of which had problems with previous commits and were reverted, while others were refactored to bring the assertions together. In addition, even if the assertions are not increased or decreased, there are commits that change to more readable assertion descriptions or rewrite to new assertion APIs, and it was found that modifying only assertions in a test method is a use case that exists well in real-world software development.

Meanwhile, there is a large proportion of commits for increasing the number of assertions. However, unlike in the case of assertion reduction, it is not known what kind of assertions will increase at the time of mutant reduction, so existing techniques, including the proposed method, cannot select the mutants to be reduced according to their impact.

6.6.2 Execution Time Optimization

In the evaluation of the proposed method, it is found that the execution time can be reduced to approximately 60%. In other words, when making choices for a mutant set that is unknown to be killed, we believe that making choices of that magnitude when the emphasis is on reliability is also important in preventing excessive reductions. Meanwhile, the algorithm for mutant selection was designed using the greedy method, so it is not an optimal solution that minimizes the execution time. It is necessary to consider the use of solvers that can handle constraints that make the error set distinguishable, thereby reducing the problem of exponential-order computational complexity and finding the optimal solution.

In terms of the execution time of the optimization itself, the execution time of error-oriented optimization is longer than that of test-oriented optimization because the number of errors is generally much higher than the number of test cases, while as an algorithm, the execution time is almost the same for the greedy method. In this experiment, the optimization time for all 53 repositories was approximately 4 hours and 16 minutes for error-oriented optimization and 1 minute and 46 seconds for test-oriented optimization, which is a significant difference. However, the mutant optimization in this study is intended to find a model to reduce the execution time of the mutation analysis, and it is done in advance of the mutation analysis, i. e., basically, the mutant optimization and the execution of the mutation analysis are done in different phases. Therefore, the execution time shown in Subsection 6.5.4 does not include the time required to execute the optimization. In both cases, most of the time taken was for mutant minimization by the greedy method, and there was little computational cost for mutant weighting.

6.6.3 Reducing Mutation Score Discrepancy

It is found that the proposed method reduces the discrepancy of the mutation score compared to the existing method. Our experiments show that the weighting to reduce discrepancy is an effective method not only for error-oriented optimization but also for test-oriented optimization. It can also be observed that there is a large difference between the proposed method and the existing method in the descending order of the average number of assertions per test case shown in Table 6.5.

While the 53 repositories used in the evaluation of execution time were systematically selected, it is possible that the three repositories used in the subsequent evaluation of errors contained bias. This is because we selected three cases based on the applicability of the tool we created for test code statement reduction and the fact that there are not many modifications before the build is passed after statement reduction. In addition, because the evaluation with varying statement reduction rates is limited to jsoup, it is necessary to evaluate the differences for more repositories in the future. In addition, it may be necessary to evaluate by using commits where the number of assertions is reduced in the project used in Subsection 6.5.4. However, in typical commits, modifications are often made to the source code in addition to assertions, which means that mutation score changes due to factors other than assertions, making it difficult to simply compare differences in mutation scores, so in this paper, we used artificial assertion reduction for evaluation.

6.6.4 Reduction Per Mutation Operator

The number of mutants per mutation operator was confirmed experimentally, and there was no significant difference in distribution between before and after the mutant reduction. We also examined the rate of reduction per mutation operator, but none of the mutation operators had distinctive reduction rates. In other words, we consider that our reduction model does not significantly reduce the mutants of a particular mutation operator, but rather it shows that it is difficult to reduce from the tendency of the mutation operator if we are considering reducing for a mutant set that is unknown to be killed. It is possible that investigating other trends in mutants in the current reduction model may contribute to the reduction of the mutant set that is unknown to be killed, and further investigation

is needed.

6.7 Summary

In this chapter, we addressed the need for a mutant reduction model that can measure the same fault detectability as the original set of mutants, even when there is a change of statements in the test code. We then proposed a new mutant reduction model, where we select mutants whose errors during test execution remain distinguishable and consider other mutants to be redundant. Furthermore, to reduce the difference in the mutation score calculation due to mutant reduction, a method was proposed to calculate the mutation score by adding the impact of the reduced mutants to the selected mutants as weights. The speed evaluation of the OSS for 53 projects reduced the execution time to approximately 40%, and the invariance of the fault detectability for three projects was evaluated with less difference than the existing test case-based mutant reduction model.

Further project investigations and reduction of the mutant set that is unknown to be killed, using the measurement information are required.

Table 6.4: Repositories used for evaluation and their LOC

Repository	Lines of Source Code	Lines of Test Code
msgpack-java	13,598	28,243
vertx-jersey	1,936	1,854
samoa	16,841	249
owner	2,866	5,520
redline-smalltalk	5,648	451
geometry-api-java	57,497	18,619
vectorz	39,793	8,439
webcam-capture	13,659	745
scribe-java	2,794	2,549
gson-fire	1,535	1,312
jsr354-api	2,319	3,487
p2-maven-plugin	1,690	153
jackson-annotations	1,471	331
auto	8,710	11,808
rxjava-jdbc	3,735	3,081
okio	3,998	4,499
jInstagram	4,015	6,748
javapoet	2,994	4,450
gwtbootstrap3	13,560	406
jphp	43,717	4,441
javaparser	13,588	3,379
JsonPath	3,765	2,714
minimal-json	1,715	4,589
jackson-core	21,451	11,131
rest-driver	3,194	5,108
retrofit	5,046	8,361
zt-zip	4,064	2,220
maven-git-commit-id-plugin	2,883	2,015
linq4j	14,307	3,979
RoaringBitmap	9,267	7,092
Ektorp	11,079	5,876
jsoup	10,696	5,188
moshi	4,136	5,515
http-request	1,391	2,721
slf4j	8,184	4,288
twilio-java	12,835	4,230
graphhopper	22,544	7,549
cassandra-reaper	5,663	1,711
LittleProxy	4,094	4,550
hbc	3,588	1,829
jsprit	20,486	16,607
wire	8,462	27,743
HikariCP	4,135	5,155
java-object-diff	5,817	1,913
docker-maven-plugin	6,085	1,924
stream-lib	4,689	3,619
jsondoc	3,841	3,543
metadata-extractor	18,731	2,344
jOOQ	130,053	1,464
pebble	6,324	4,725
alf.io	7,226	499
zxing	35,164	7,582
Algorithms	1,016	1,356

Table 6.5: Numbers of test cases and assertions in jsoup, zt-zip, and jInstagram

	# of test cases	# of assertions	average # of assertions per test case
jsoup	538	1585	2.946
zt-zip	6	243	40.5
jInstagram	547	748	1.367

Table 6.6: Execution time of mutation analysis

	Total execution time (msec)	Ratio to total execution time without optimization
No optimization	814,090,026	100%
Error-oriented optimization	333,001,232	40.9%
Test-oriented optimization	182,450,441	22.4%

Table 6.7: Number of killed mutants in reduction of assertions

		All mutants	Without weighting	With weighting
jsoup	No optimization	9,096	3,032	3,032
	Error-oriented opt.	5,915	2,265	3,054
	Test-oriented opt.	2,316	1,039	3,181
zt-zip	No optimization	2,173	1,696	1,696
	Error-oriented opt.	1,196	1,110	1,700
	Test-oriented opt.	296	258	1,845
jInstagram	No optimization	1,080	92	92
	Error-oriented opt.	795	56	92
	Test-oriented opt.	446	48	90

Table 6.8: Number of killed mutants in reduction of test code's statements in jsoup

	All mutants	20%	40%	60%	80%
No optimization	9,096	8,807	7,747	6,274	3,032
Error-oriented opt.	5,915	8,819	7,745	6,262	3,054
Test-oriented opt.	2,316	8,865	7,808	6,327	3,181

Chapter 7

Related Work

7.1 Speeding Up Mutation Analysis

As mentioned in Chapter 1, the techniques to reduce the computational cost of mutation analysis are classified into "do fewer," "do smarter" and "do faster." In this section, we introduce the related studies of execution cost reduction ("do smarter" and "do faster").

Originally proposed by Howden [95], *Weak mutation* is one of "do smarter" approaches, which is an approximation technique that compares the internal states of the mutant and the original program immediately after execution of the mutated position of the program. *Split-stream execution* is another "do smarter" approach, which is introduced in Section 4.2.3 and is adopted in our research. Durelli et al. [67] implemented split-stream execution and compares program states for weak mutation. It splits the execution by methods unlike our approach which splits by instruction. Papadakis and Malevris [162]'s approach is quite similar to split-stream execution in the point of view of state branching at mutation location. They utilized *Dynamic Symbolic Execution* (DSE) and MSG to run mutation analysis and generate test data based on the mutants. The technique by Just et al. [105] is also "do smarter" approach. The technique avoids unnecessary infected states and reduces mutation analysis time by 40% on average.

As "do faster" approach, Ma et al. [130] adopts MSG for generating behavioral mutants that change the behavior of the program, e.g., overriding method calling, overloading method, and *bytecode translation* for generating structural mutants that change the structure of the program, e.g., inherited variables, access modifier. These techniques are showed in Section 4.2.1 and 4.2.2 respectively. Our metamutation approach is originally derived to MSG in order to identify source-level program elements for analyzing bitcode-level representation.

7.2 Mutants Optimization

Various mutant reduction methods (called "do fewer" approach) have been proposed in the past.

Some papers compare mutant reduction methods due to selection of mutation operators with those due to random sampling. Budd [32] and Acree [1] showed that sampling 10% of mutants can approximate the original mutation score with 99% accuracy. Wong et al. [205] compared leaving a certain percentage of mutants for each mutation operator versus using only two mutation operators and achieved comparable mutation scores and comparable accuracy in both. Zhang et al. [209] compared operator-based reduction methods with random sampling and showed that random sampling was superior.

There are a number of studies that examine a sufficient set of mutation operators. Offutt et al. [157] showed that mutation scores can be measured with six different mutation operators with an accuracy of 99.5% of the original. Barbosa et al. [19] gave insight into mutation operator selection, achieving a mutation score of 99.6% of the original accuracy by mutants reduced to 65.02%.

Deng et al. [56] demonstrated the effectiveness of statement deletion mutation operators for test evaluation in Java. Delamaro et al. [51] found that a single statement deletion mutation operator had a similar effect as when all operators were used. They measure the effect of the statement deletion mutation operator on mutation analysis, but not on fault localization.

Another "do fewer" approach is detecting equivalent mutants and duplicated mutants. Papadakis et al. [163, 114] propose Trivial Compiler Equivalence technique that declares equivalences only for those mutants which their compiled object code is identical to the compiled object code of the original program. Kintis and Malevris [115, 113, 116] devised a method to detect a large portion of equivalent mutants by using data flow patterns through static analysis.

These mutant reduction methods are heuristics for mutants that are not known to be killed, and do not represent a limit on the amount of mutant reduction.

There is a test case-based reduction model of Gopinath et al. [86] for the limits of mutant reduction. Gopinath et al. presented a theoretical and an experimental upper bound on the mutant reduction with random sampling. The theoretical upper limit of the reduction is 58.2% under simplifying assumptions of uniform redundancy of faults in mutants, while the experimental upper limit is 13.078% on average.

To the best of our knowledge, the mutation score for a set of mutants after the reduction has not been mentioned in any existing studies.

7.3 Mutation-based Fault Localization

In Section 5.2.3, we introduced MUSE and Metallaxis as major methods of MBFL, but there are some more improved methods based on them.

Li and Zhang [124] proposed TraPT, a method that utilizes the impact information of each mutant on each assertion to compute more precise fault localization information. In experiments with Defects4J, TraPT showed a higher fault localization performance compared to the existing MBFL.

Gong et al. [84] proposed a method called Dynamic Mutation Execution Strategy (DMES), which dynamically select mutants and test cases to reduce the execution cost of MBFL. Initially, the upper limit suspicion value for each mutant is calculated for the failure test case only, and only mutants whose value is above a certain value are selected. Furthermore, during the execution of a mutant's successful test cases, if the number of successful test cases that can be killed exceeds a certain percentage, the execution of the remaining successful test cases in that mutant is skipped.

Lôbo de Oliveira et al. [127] proposed a method, FTMES, which generates only mutants in the places where the failed test case executes, noting that the successful test case does not contribute significantly to fault localization. The evaluation using Defects4J shows that it is faster and more accurate than existing methods such as DMES.

The method described here can be adopted even when mutation operators are limited, so it may be used to improve SDL-MBFL.

7.4 Industrial Case Studies of Mutation Analysis

While there are still many obstacles to practical use of mutation analysis, several case studies of industrial applications have been published.

As a first case study of mutation analysis, Daran [49] analyzed the error trends of 12 faults and 24 mutants in a nearly 1,000-line C program created by students from specifications in the nuclear industry. The results showed that 85% of the errors made by the mutants were the same as the actual errors.

Baker and Habli [18] applied mutation analysis to two safety-critical software systems on aircraft, written in C and Ada. In their experiments, they were able to find a subset of effective mutation operators and detected shortfalls that could not be detected by the test suites built with processes and coverage criteria that meet the requirements of existing standards.

Možucha and Rossi [143] conducted an experimental evaluation of a mutation analysis tool for Java and concluded that what is practically necessary is to change the settings that affect performance from the default settings.

Ramler et al. [177] carried out a mutation analysis on about 60,000 LOCs of embedded software for safety-critical machine control. The results of their mutation analysis suggest a deficiency of test cases that satisfy 100% MC/DC coverage and provide a valuable guide to improvement. The execution time was over 4,000 hours and all the effort took about half a person-year.

Petrovic et al. [171, 170] obtained three lessons learned through the implementation case of large-scale mutation analysis at Google. The first lesson is that besides equivalent mutants and redundant mutants, "unproductive mutants" that are practically useless force developers to waste their time. Another lesson is that while it is typical that mutation scores are calculated at the method or file level, developers want to measure test sufficiency at the commit level. The last lesson is that developers only need to make their test suites better, and contrary to researchers' beliefs, mutation adequacy is not cost-worthy.

Delgado et al. [53] presented a case study of applying mutation analysis to 15 functions of C in the mission-critical domain, nuclear industry. They report that selective mutation allowed them to find mutants that could not be killed by existing branch-covered test cases, and that trivial compiler equivalence (TCE) significantly reduced the number of equivalent mutants.

7.5 Evaluation of Debugging Techniques for Industrial Software

Siemens suite is the most frequently used industrial software for the evaluation of fault localization and has been used in more than 90 papers according to [204]. However, it is not so large in scale and employs artificially inserted defects, rather than actual defects, as the evaluation target.

There are several studies on the application of automatic program repair techniques to industrial software. Naitou et al. [144] reported that one out of nine faults could be fixed automatically by applying jGenProg, an automated program repair technique, to enterprise-developed Java programs and faults. Ikeda et al. [214] tried using Prophet, an automatic program repair technology, against actual C programs and faults in a company, and reported that one patch was obtained out of two. Noda et al. [152] improved 2 out of 20 successful patch generation with Elixir, an automated program repair technology, in Java software that has been developed and operated in the enterprise for more than 13 years, resulting in 8 out of 20 successful patch generation. In these studies, the fault localization

used in the automatic program repair is SBFL by Ochiai, and the evaluation of the fault localization itself is not described.

7.6 Applications of Mutation Analysis

The results of the mutation analysis allow for the generation and selection of test oracles. Fraser and Zeller [76, 77] check each output of the program to produce an assertion of the output that distinguishes the mutant from the original program. Staats et al. [185, 78] rank the variables with high efficiency in finding embedded faults in mutation analysis and generate their assertions.

Test data generation is another powerful application of mutation analysis. Constraint-based test generation is a method first proposed by Offutt [159] to generate killable test data for each mutant by giving the solver the conditions of the test data to kill the mutant as constraints. A similar approach, combining symbolic execution and mutation analysis, was originally proposed by Papadakis et al [166, 162]. The idea is to generate a test by symbolic execution to get the mutant infection condition in the mutant schema function.

Our another tool, called Phanta, is test code quality measurement tool leveraging mutation analysis. Figure 7.1 shows Phanta's GUI. It can monitor the three metrics, fault detectability, maintainability, and speed. It leverages mutation analysis in terms of fault detectability and maintainability. For fault detectability, Phanta simply uses mutation analysis for measuring mutation score. For maintainability of test code, Phanta incorporates two analyses. The first is *active assertion analysis* that flags assertions which don't kill any mutants as essentially redundant, and hence candidates for removal, in order to improve test code readability. Multiple assertions could adversely influence maintainability [20]. Phanta is applied to an actual development project related to factory automation. Through the application, the following lessons are derived.

- Make the report understandable for developers to lead them to the next action
- Make the tool flexible and customizable for selecting the target to be measured
- Automated test code refactoring could be helpful to get more maintainable test code

7.7 Tools for Mutation Analysis

Although there are implementations of mutation analysis in various languages not only for programming but also specification and modeling, the most common ones are for C/C++ and Java. [164] introduces 76 mutation analysis tools in various languages, 27 of which are for C/C++ or Java, while at most 3 of them are for each of the other languages.

Table 7.1 shows a comparison of Mutation analysis tools for C/C++. No particular checkmark indicates that only source-level mutations, method-level mutation operators, and first-order mutations are supported.

Implementations of the muntant schemata generation have been around for a long time, TUMS and Plextest being examples. On the other hand, bitcode-level mutation has been implemented more and more in recent years with the spread

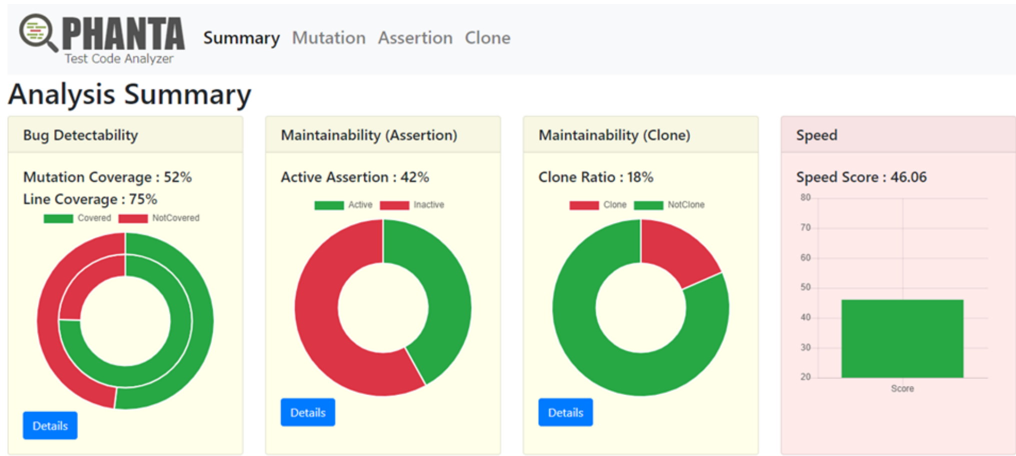

Figure 7.1: Phanta: A Test Code Quality Measurement Tool

Table 7.1: A comparison of Mutation analysis tools for C/C++ (OO:Object-oriented mutation operators, BM:Bitcode-level mutation, MSG:Mutant schemata generation, HOM:Higher order mutation, SSE:Split-stream execution)

Name	Year	OO	BM	MSG	HOM	SSE	Publicly available	Open sourced
TUMS [193, 195, 194]	1995			✓				
Proteum [52]	2001						✓	✓
Plextest [97]	2005			✓				
MutGen [11, 10]	2003							
ESTP [71]	2008							
Milu [101]	2008				✓		✓	✓
SMT-C [48]	2012							
CCMutator [119]	2013				✓		✓	✓
llvm-mutate [182]	2013		✓				✓	✓
MuVM	2016		✓	✓	✓	✓		
Mutate++ [128]	2017						✓	✓
MuCPP [54]	2017	✓					✓	✓
AccMut [198]	2017		✓	✓		✓	✓	✓
Mull [57]	2018		✓				✓	✓
MUSIC [172]	2018						✓	✓
Dextool [28]	2018						✓	✓
SRCIROR [90]	2018		✓				✓	✓
Mart [37]	2019		✓				✓	✓

of LLVM, such as Mull [57], SRCIROR [90] and Mart [37]. MuCPP [54] is the only tool that supports object-oriented mutation operators.

Table 7.2 also shows a comparison of Mutation analysis tools for Java. Again, no particular checkmark means that it only supports source-level mutations, method-level mutation operators, and first-order mutations.

There are many tools for other languages as well. For C#, ILMutator [60] and Stryker.NET [150] can perform mutation analysis on programs running on the .NET CLI. For JavaScript, Mutandis [139, 140], AjaxMutator [151] and Stryker [149] supports JavaScript-specific mutant operators. For Python, MutPy [59]

Table 7.2: A comparison of mutation analysis tools for Java (OO:Object-oriented mutation operators, BT:Bytecode translation, MSG:Mutant schemata generation, HOM:Higher order mutation, Concurrency:Concurrency-related mutation operators)

Name	Year	OO	BT	MSG	HOM	Concu-rrency	Publicly available	Open sourced
Jester [142]	2001						✓	✓
JavaMut [43]	2002	✓						
MuJava [130]	2004	✓	✓	✓			✓	
ByteMe [65]	2006		✓					
Jumble [192]	2007		✓				✓	✓
Javalanche [181]	2009		✓	✓			✓	✓
PIT [44]	2010		✓				✓	✓
MuTMuT [82]	2010					✓		
Judy [131]	2010	✓		✓			✓	
Bacterio [135]	2010		✓	✓			✓	
MAJOR [106]	2011						✓	
Paraμ [132]	2011	✓				✓		
Comutation [83]	2013					✓		
HOMAJ [160]	2014				✓			
LittleDarwin [168]	2017				✓		✓	✓

supports higher order mutation and objected-oriented mutation operators, while Cosmic Ray [14] and Mutmut [94] are focused on practical use. For Ruby, a tool named mutant [123] is the only one available.

7.8 Data flow Analysis for Testing and Debugging

Mutation analysis, as described in Chapter 2, checks whether the execution of the test case reaches the mutation point, whether the execution state is infected with an abnormality due to the injected artificial fault, whether the abnormal state is propagated through the execution after the mutation location, and finally whether the test oracle can detect the abnormality of the state. In other words, we artificially change the data and check whether the effect of the change is propagated to the output as test oracle.

Measuring the impact of certain program elements is also done in the field of program analysis, such as data flow analysis, information flow analysis, taint analysis, and so on. Data flow analysis [6] has been studied as a program optimization technique in compilers. By analyzing the data dependencies between the definition and use of variables, it is used to optimize the program by removing useless assignment instructions.

Information flow analysis [58, 179] is a method to analyze the dependency between input and output using to determine the leakage of information that should be kept secret. While data flow analysis analyzes all data dependencies, information flow analysis is sufficient to analyze only the dependencies from the input of the information to be kept secret.

Taint analysis [147] is a method of highlighting security risks by marking data that may have been given by a malicious user as tainted data and tracking which variables propagate during program execution. This technique has many similarities with information flow analysis, although the application is different,

and only analyzes the dependence from the input.

A method of using data flow analysis for test design, called data flow testing [72, 161], was proposed. This is a method to design or generate tests using the coverage criterion of how well the test can cover the data definition and use pairs. There are several empirical studies that have compared data flow testing and mutation testing. Mathur and Wong [136] concluded that mutation-based criteria are more difficult to satisfy than all-use criteria. The experimental results of Offutt et al. [153] showed that while both methods were effective, the mutation-adequate test set was closer to satisfying the dataflow criterion and was able to detect more defects. The experiment by Frankl et al. [73] summarized that the mutation testing was more effective in five out of nine subjects, the all-use criterion was effective in two subjects, and there was no clear winner in two subjects. Kakarla et al. [107] conducted a meta-analysis of these comparative studies of mutation testing and data flow testing and concluded that mutation testing is at least two times more effective than data flow testing, but three times less efficient.

Some works use data flow testing for fault localization. A method proposed by Agrawal et al. [5] is based on the assumption that the fault lies in the slice of the test case that fails at runtime, rather than succeeding at runtime. As a result, developers can focus the statements on the failed slices. Santelices et al. [180] proposed a lightweight fault localization technique that uses coverage criteria such as statements, branches, and def-use pairs to detect suspicious statements in a program.

Program slicing is a technique that focuses on the dependencies between statements in a program, such as data flow and control flow, and extracts a statement set that have dependencies with a specified statement, called program slice. Although program slicing has a variety of applications, it can be used to improve debugging efficiency by narrowing it down to only the part related to the statement in which the error was caught, such as an assertion. Slicing techniques can be roughly classified into two categories: static slicing and dynamic slicing. Static slicing proposed by Weiser [200] creates a program dependency graph that combines control dependency and data dependency for the target program, and extracts dependent statements by tracing the graph backwards from the target statement. Dynamic slicing, proposed by Agrawal et al. [4], builds a program dependency graph by recording dependencies when actually feeding input to the program and executing it, and extracts dependent statements by tracing the graph backwards from the target statement as in static slicing. Since static slicing does not require any input data, it is generally low overhead. However, the slice size tends to be large, and in extreme cases, the entire source code is extracted as a slice.

Both mutation analysis and data flow analysis can be used to examine how a given program location impacts the output, but the biggest difference is how the impact on the output is measured. Mutation analysis introduces changes to program elements and examines whether those changes affect the output of the program. Data flow analysis, on the other hand, records dependencies and extracts statements that have dependencies on the output.

Missing impact chains in the analysis such as dependencies and propagation of abnormal state can be present in both. In the case of mutation analysis, even if a change is made to a statement that has dependencies in the output, the change may not necessarily show up in the output depending on the test data and mutation operator. In data flow analysis, dependencies on the output will not be missed if only the source code is traced, but if native code or database

access is in the middle of the analysis, dependencies may not be tracked.

Also, the execution cost of both methods is CPU intensive. Data flow analysis records and analyzes all dependencies for each statement. Mutation analysis, on the other hand, requires running as many tests as there are variations in the changes.

This difference in nature leads to a difference in usage. Mutation analysis is suitable when you want to analyze the entire system, including external environment and libraries, while data flow analysis is suitable for module and file analysis.

Chapter 8

Conclusion

8.1 Summary

In industrial software, the impact of economic losses due to faults on the world is significant, and this trend has been increasing in recent years. Detecting and fixing faults is a highly time-consuming process, and testing and debugging techniques are demanded to support it. Mutation analysis is expected to be a powerful foundation for test and debug.

In this thesis, we addressed the challenges of computational cost in mutation analysis with three approaches: "do fewer", "do faster", and "do smarter". For the challenge of "do faster", the mutation location in bitcode translation is lost by compile optimization. For the challenge of "do smarter", SSE is not applicable for naive compiler-based execution methods due to branching the execution stream. For the challenge of "do fewer", a test case-based mutant reduction model may over-reduce mutants and fail to measure accurate fault detectability.

First, we presented MuVM as a "do faster", and "do smarter" approach, a tool for fast higher-order mutation analysis using four techniques (Chapter 4). *Metamutation* prevents the loss of mutation locations in bitcode-level at compile time by replacing the program elements to be mutated with metamutation functions. *Mutation on virtual machine* interprets bitcode containing metamutation functions and performs mutation analysis on the VM. *Higher order split-stream execution* splits the execution state into mutated and unmutated states at the mutation location during execution, saving the common execution cost to the mutation location. *Online adaptation technique* reduces the cost of running unwanted mutants by dynamically creating a mutant running state. The experiments showed that the execution time was significantly shorter in MuVM compared to the existing methods.

Second, we proposed and evaluated a mutation-based fault localization using statement deletion mutation as an application of MuVM (Chapter 5). The subject of the evaluation was a system reengineering project in an enterprise, where incompatibilities before and after the renewal were treated as faults, and nine actual faults were localized at a time. In a comparative experiment of the formulas for each suspiciousness, Tarantula showed the best fault localization performance. In a comparative experiment of mutation operators, it was found that the fault localization performance was significantly different with and without SDL (statement deletion).

Finally, as a "do fewer" approach, we proposed a new mutant reduction model that focuses on the errors caused by mutation (Chapter 6). We also proposed a method of weighting the remaining mutants according to the reduced mutants with the same error. Our experiments revealed that this mutation reduction

model and weighting allows us to measure accurate mutation scores compared to an existing method, the test case-oriented mutation reduction model. In addition, we show that our mutation reduction method can reduce the execution time of mutation analysis by about 40% in 53 OSS projects.

This thesis showed that these techniques can assist in efficient mutation analysis for real-world software, including industrial use.

8.2 Overall Evaluation

This section presents an overall evaluation of this thesis in terms of coverage measurement and fault localization.

8.2.1 Overall Evaluation as Coverage Measurement Technique

To conclude this thesis as a coverage measurement technique, we compare it with the state-of-the-art (SoTA) technology using the quality characteristics of coverage measurement required by industry from the results of the systematic literature review presented in Chapter 3.

Speed

Our method Our techniques accelerates high order mutation analysis by 10 to 10,000 times by reducing the number of compilation and execution instructions. In addition, we proposed a model that can reduce mutants by 60% while enabling measurement of accurate fault detectability. Note that this mutant reduction model does not directly contribute to the actual mutant reduction, and the speedup effect of combining these techniques is unknown.

SoTA methods There is no one that combines the same techniques as ours; AccMut [198] is a later technique than MuVM but closest to it, and AccMut is more than 10 times faster than MuVM for "tcas" program.

Usability

Our method Although not the subject of this paper, we developed Phanta, a platform for analyzing code from multiple perspectives. The report of the mutation analysis results needed to be easy to understand so that the developer's next action would be clear.

SoTA methods Although the state of the art in usability efforts as research is not clear, PIT is preferred by many practical users.

Accuracy

Our method Our optimization model retains almost 100% accuracy compared to the original mutation score, but the accuracy in actual mutant reduction is unknown because it has not been measured for a mutant set that is unknown to be killed.

SoTA methods There is a model that can reduce mutants by about 13% while maintaining accuracy, but it determines redundancy at the test case level, which is more coarse-grained than our method at the statement level.

Scalability

Our method We tried higher order mutation analysis on a program with about 6,000 lines. Further evaluation on a larger program is needed.

SoTA methods AccMut [198] performs mutation analysis on a large scale software with 42KLOC and over 5,000 test cases. In an industrial case, Ramler et al. [177] applied a mutation analysis to about 60,000 LOCs of embedded software in safety-critical areas.

Memory consumption

Our method SSE, which is incorporated in our method, requires a lot of memory to maintain the execution state. Higher order mutations are even more so.

SoTA methods The memory consumption of individual tools is unknown, but those that use internal state, such as SSE, have the same issues.

Reliability

Our method Since our software is not yet mature, we cannot say that it is highly reliable. We expect to improve its reliability through future experience.

SoTA methods As a tool, PIT is the most widely used and reliable tool.

This thesis has contributions on methods to improve speed and accuracy, and especially has advantages in higher order mutation against state-of-the-art technology. However, we believe that improvements are needed in terms of scalability and memory consumption.

8.2.2 Overall Evaluation as Fault Localization Technique

As shown in Chapter 5, Kochhar et al. [118] surveyed 386 practitioners about their expectations of defect localization and then examined the state-of-the-art for seven factors they considered important. From the seven factors including "availability of debugging data", "granularity level", "minimum success criterion", "Scalability", "Efficiency", "Rationale", and "IDE Integration", we list the perspectives required by industry in fault localization and compare our method with the state-of-the-art. "Availability of debugging data" means which is assumed to be available by prior fault localization studies. "Granularity level" is the level of granularity used to pinpoint defects such as classes, methods, and statements. "Trustworthiness" refers to how high the actual fault ranks among the ranked program elements. The indicator here is the percentage of faults included in the top 5, which are the most preferred in the survey results. "Scalability" refers to the size of the target program, and "efficiency" refers to how long it takes to complete fault localization. "Rationale" is why some program locations are marked as suspicious. "IDE Integration" means that fault localization is integrated into the IDE.

Availability of debugging data

Our method We only use test data as our debug data, which is the most commonly available data.

SoTA methods Most of the existing techniques use test cases as debugging data, followed by bug reports.

Granularity level

Our method Our method supports fault localization at the statement level of granularity, which is the finest granularity.

SoTA methods Only two papers [121, 206] work at method level granularity which is the most preferred option, and most papers work at statement level granularity which is second most preferred option.

Trustworthiness

Our method Applying our tool to a real-world program that contains multiple faults across multiple lines at the same time, we found that the percentage of faults in the top-5 of the suspiciousness rank is 33%.

SoTA methods None of the papers can satisfy 75% of success rate in the top-5 position, while 5 papers (e.g. [102, 110, 178]) can satisfy 50% of the success rate. Unfortunately, those papers work at a coarser level of granularity (e.g. class level) that is not preferred in the survey.

Scalability

Our method We performed fault localization on the 13KLOC source code in real server monitoring.

SoTA methods Techniques in 6 papers (e.g. [134, 121, 206, 102]) support at least 100KLOC and 7 [13, 15, 174] support at least 10KLOC, that can satisfy at least 75% and 50% of the survey respondents, respectively.

Efficiency

Our method Since our method is based on mutation analysis, it is less efficient than the usual spectral-based fault localization and took about 150 hours.

SoTA methods 5 papers (e.g. [13, 121, 178]) shows their techniques produce output in less than a minute, that satisfy 90% of the survey respondents.

Rationale

Our method Our method, like many other methods, only highlights faulty program elements.

SoTA methods 2 papers [134, 187] provide a graph-based structure that a practitioner can inspect to better understand why the elements are highly ranked as faults.

IDE Integration

Our method Our fault localization tool doesn't support IDE integration.

SoTA methods There is no work that is integrated into IDE. The closest is the work by Zhou et al. [210], which has been integrated into Bugzilla, however, Bugzilla is not an IDE.

This thesis showed that our fault localization method is superior in terms of granularity and trustworthiness, but there are some issues in terms of efficiency, so it is necessary to incorporate the method into continuous integration for practical use.

8.2.3 Overall Evaluation with Future Prospects

To the best of our knowledge, AccMut and other accelerators in SoTA do not support higher order mutation. If we apply the state reduction technique in AccMut to higher order mutation, we need to consider handling multiple mutation descriptions in the state, one of which is our other technique [190]. Although the issue of execution cost still remains for higher order mutation, in the future, new techniques such as state reduction and more lightweight and optimized implementations of virtual machines may enable higher order mutation to be used in larger scale software. If further speed-up of higher order mutation can be expected, it may be effective in automated program repair and fault localization for multiple locations. Existing study [152] has shown that there is a need for correction of multiple faults rather than a single fault in industry. higher order mutation can be expected to be used in the automation of testing and debugging for complex faults, as actually required in industry.

8.3 Future work

8.3.1 Further Improvement in Performance

Reduction of the Unknown Set of Mutants

While this thesis presents a novel mutant reduction model, there is a future challenge for reduction of mutant set that is unknown to be killed, such as those in new programs that have not yet been analyzed. One direction of reduction is the selection of mutation operators. However, we did not find the trend in our experiment. As powerful machine learning approaches have become more readily available in recent years, one idea is to automatically reduce mutants from the large amount of various data generated by past mutation analysis.

Test Case Selection

While this thesis focuses on reducing the time spent in compiling and testing during mutation analysis, it does not incorporate methods to reduce the number of test cases to run. Whereas there is a method to select only the test cases that cover the mutation locations of each mutant, one of MuVM's techniques, Online adaptation, dynamically selects only the necessary mutation locations at the time of execution of each test case, so the effect is practically the same. However, in the mutation-based fault localization technique, it is not necessary to run many passed test cases because the information in the failed test cases is more important. It is expected to shorten execution time by identifying necessary passed test cases and reducing unnecessary ones.

8.3.2 Other Practical Issues

Elimination of Equivalent mutants

Although equivalent mutants were considered out of scope in this thesis, they are important issues in practical terms. Equivalent mutants not only prevent accurate measurement of fault detectability by mutation analysis, but they can also result in wasted execution time and wasteful work by developers that cannot be used to improve the test suite. The detection of equivalent mutants is difficult as it has proven to be an undecidable problem, but we expect that the various methods that have been proposed in academia will help to solve it in industry.

Finding Subtle Higher Order Mutants

Our tool MuVM can generate HOMs, but not all of them are considered useful, and finding subtle HOMs, which are harder to kill and more valuable for finding faults, will help us to build a more powerful test suite. The detection of subtle HOMs is very computationally expensive and needs to be solved by utilizing various search methods proposed in the academic community.

8.3.3 Towards Further Industrial Adoption

Other Applications of Mutation Analysis

In this study, we implemented and evaluated mutation-based fault localization as an application of mutation analysis. There are other applications that use mutation analysis as a basis. Test case generation, combined with techniques such as symbolic execution, generates test cases that can kill more mutants. In automated program repair, the mutation operators that do the transformation of a program element into a faulty one are regarded as operations that remove a fault from a faulty program element, and find mutants that make all test cases pass, including those that had failed due to the fault. Test oracle selection considers test oracles that can kill many mutants in mutation analysis to be useful and reduces unnecessary test oracles to increase the maintainability of the test suite. The implementation of these applications is a future challenge.

Further Experiments with Industrial Software

In this thesis, we have evaluated industrial software in C, but we have not evaluated industrial software in other languages such as Java. We expect that more diverse and realistic evaluations in industrial software will contribute to the further development of mutation analysis techniques.

References

[1] Allen Troy Acree Jr. “On Mutation”. PhD thesis. Atlanta, GA, USA, 1980.

[2] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. “Code coverage analysis in practice for large systems”. In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 736–745.

[3] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. *Design of Mutant Operators for the C Programming Language*. Tech. rep. SERC-TR-41-P. West Lafayette, Indiana: Purdue University, Mar. 1989.

[4] Hiralal Agrawal and Joseph R Horgan. “Dynamic program slicing”. In: *ACM SIGPlan Notices* 25.6 (1990), pp. 246–256.

[5] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. “Fault localization using execution slices and dataflow tests”. In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE. 1995, pp. 143–151.

[6] Frances E. Allen and John Cocke. “A program data flow analysis procedure”. In: *Communications of the ACM* 19.3 (1976), p. 137.

[7] Domenico Amalfitano, Vincenzo De Simone, Anna Rita Fasolino, and Vincenzo Riccio. “Comparing model coverage and code coverage in model driven testing: an exploratory study”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE. 2015, pp. 70–73.

[8] P. Ammann, M. E. Delamaro, and J. Offutt. “Establishing Theoretical Minimal Sets of Mutants”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 21–30. DOI: 10.1109/ICST.2014.13.

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd. USA: Cambridge University Press, 2016. ISBN: 1107172012.

[10] J. H. Andrews, L. C. Briand, and Y. Labiche. “Is Mutation an Appropriate Tool for Testing Experiments?” In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ' 05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 402–411. ISBN: 1581139632. DOI: 10.1145/1062455.1062530. URL: https://doi.org/10.1145/1062455.1062530.

[11] J. H. Andrews and Yingjun Zhang. “General test result checking with log file analysis”. In: *IEEE Transactions on Software Engineering* 29.7 (2003), pp. 634–648.

[12] Andrea Arcuri. "RESTful API automated test case generation". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, pp. 9–20.

[13] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. "Fault localization for dynamic web applications". In: *IEEE Transactions on Software Engineering* 38.2 (2011), pp. 314–335.

[14] Sixty North AS. *Cosmic Ray: mutation testing for Python*. https://cosmic-ray.readthedocs.io/. 2017.

[15] George K Baah, Andy Podgurski, and Mary Jean Harrold. "Mitigating the confounding effects of program dependences for effective fault localization". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 146–156.

[16] Thomas Bach, Artur Andrzejak, and Ralf Pannemans. "Coverage-based reduction of test execution time: Lessons from a very large industrial project". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2017, pp. 3–12.

[17] Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo. "The impact of coverage on bug density in a large industrial software project". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2017, pp. 307–313.

[18] Richard Baker and Ibrahim Habli. "An empirical evaluation of mutation testing for improving the test quality of safety-critical software". In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 787–805.

[19] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. "Toward the determination of sufficient mutant operators for C †". In: *Software Testing, Verification and Reliability* 11.2 (2001), pp. 113–136. DOI: 10.1002/stvr.226. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.226.

[20] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. "An empirical analysis of the distribution of unit test smells and their impact on software maintenance". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Sept. 2012, pp. 56–65. DOI: 10.1109/ICSM.2012.6405253.

[21] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. "Are test smells really harmful? An empirical study". In: *Empirical Software Engineering* 20.4 (Aug. 2015), pp. 1052–1094. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9313-0. URL: https://doi.org/10.1007/s10664-014-9313-0.

[22] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-stack Research on Continuous Integration". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 447–450. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.24. URL: https://doi.org/10.1109/MSR.2017.24.

[23] Henning Bergström and Eduard Paul Enoiu. "Using timed base-choice coverage criterion for testing industrial control software". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2017, pp. 216–219.

[24] Stefan Berner, Roland Weber, and Rudolf K Keller. "Enhancing software testing by judicious use of code coverage information". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 612–620.

[25] SM Bindu Bhargavi, SB Nandeeswar, V Suma, and Jawahar J Rao. "Conventional testing and combinatorial testing: A comparative analysis". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. Vol. 1. IEEE. 2016, pp. 1–5.

[26] Johan Blom, Bengt Jonsson, and Sven-Olof Nyström. "Industrial evaluation of test suite generation strategies for model-based testing". In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2016, pp. 209–218.

[27] Fiorenza Brady. *Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year*. `http://www.prweb.com/releases/2013/1/prweb10298185.htm`. 2013.

[28] Joakim Brännström. *dextool*. `http://joakim-brannstrom.github.io/dextool/`. 2018.

[29] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. "Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers" ". In: ().

[30] Georg Buchgeher, Christian Ernstbrunner, Rudolf Ramler, and Michael Lusser. "Towards tool-support for test case selection in manual regression testing". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2013, pp. 74–79.

[31] Timothy A Budd and Dana Angluin. "Two notions of correctness and their relation to testing". In: *Acta informatica* 18.1 (1982), pp. 31–45.

[32] Timothy Alan Budd. "Mutation Analysis of Program Test Data". PhD thesis. New Haven, CT, USA, 1980.

[33] Tianqin Cai, Zhao Zhang, and Ping Yang. "Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd." In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 2020, pp. 93–96.

[34] Ryan Carlson, Hyunsook Do, and Anne Denton. "A clustering approach to improving test case prioritization: An industrial case study". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2011, pp. 382–391.

[35] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S. P. Midkiff. "Statistical Debugging: A Hypothesis Testing-Based Approach". In: *IEEE Transactions on Software Engineering* 32.10 (Oct. 2006), pp. 831–848. ISSN: 2326-3881. DOI: `10.1109/TSE.2006.105`.

[36] Peter Charbachi, Linus Eklund, and Eduard Enoiu. "Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers?" In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2017, pp. 92–99.

[37] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. "Mart: A Mutant Generation Tool for LLVM". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 1080–1084. ISBN: 9781450355728. DOI: 10.1145/3338906.3341180. URL: https://doi.org/10.1145/3338906.3341180.

[38] Boyuan Chen. "Improving the software logging practices in DevOps". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* IEEE. 2019, pp. 194–197.

[39] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jiang. "An automated approach to estimating code coverage measures via execution logs". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 2018, pp. 305–316.

[40] M-H Chen, Michael R Lyu, and W Eric Wong. "Effect of code coverage on software reliability measurement". In: *IEEE Transactions on reliability* 50.2 (2001), pp. 165–170.

[41] Mei-Hwa Chen, Michael R Lyu, and W Eric Wong. "An empirical study of the correlation between code coverage and reliability estimation". In: *Proceedings of the 3rd International Software Metrics Symposium.* IEEE. 1996, pp. 133–141.

[42] Mei-Hwa Chen, Michael R Lyu, and W Eric Wong. "Incorporating code coverage in the reliability estimation for fault-tolerant software". In: *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems.* IEEE. 1997, pp. 45–52.

[43] Philippe Chevalley and Pascale Thevenod-Fosse. "A mutation analysis tool for Java programs". In: *International journal on software tools for technology transfer* 5.1 (2003), pp. 90–103.

[44] Henry Coles. *PIT.* http://pitest.org.

[45] IEEE Standards Committee et al. "Ieee std 610.12-1990 ieee standard glossary of software engineering terminology". In: *online] http://st-dards. ieee. org/reading/ieee/stdpublic/description/se/610.12-1990 desc. html* (1990).

[46] Joao Carlos Cunha, Ricardo Barbosa, and Gilberto Rodrigues. "On the use of boundary scan for code coverage of critical embedded software". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering.* IEEE. 2012, pp. 341–350.

[47] Jacek Czerwonka. "On use of coverage metrics in assessing effectiveness of combinatorial test designs". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops.* IEEE. 2013, pp. 257–266.

[48] H. Dan and R. M. Hierons. "SMT-C: A Semantic Mutation Testing Tools for C". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* 2012, pp. 654–663.

[49] Muriel Daran. "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations". In: *In Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM Press, 1996, pp. 158–171.

[50] V. Debroy and W. E. Wong. “Insights on Fault Interference for Programs with Multiple Bugs”. In: *2009 20th International Symposium on Software Reliability Engineering*. Nov. 2009, pp. 165–174. DOI: 10.1109/ISSRE.2009.14.

[51] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt. “Experimental Evaluation of SDL and One-Op Mutation for C”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 203–212. DOI: 10.1109/ICST.2014.33.

[52] Márcio Eduardo Delamaro and José Carlos Maldonado. “Mutation Testing for the New Century”. In: ed. by W. Eric Wong. Norwell, MA, USA: Kluwer Academic Publishers, 2001. Chap. Proteum/IM 2.0: An Integrated Mutation Testing Environment, pp. 91–101. ISBN: 0-7923-7323-5. URL: http://dl.acm.org/citation.cfm?id=571305.571326.

[53] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. “Evaluation of Mutation Testing in a Nuclear Industry Case Study”. In: *IEEE Transactions on Reliability* 67.4 (2018), pp. 1406–1419.

[54] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. “Assessment of class mutation operators for C++ with the MuCPP mutation system”. In: *Inf. Softw. Technol.* 81 (2017), pp. 169–184.

[55] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer* 11.4 (Apr. 1978), pp. 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136. URL: http://dx.doi.org/10.1109/C-M.1978.218136.

[56] L. Deng, J. Offutt, and N. Li. “Empirical Evaluation of the Statement Deletion Mutation Operator”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 84–93. DOI: 10.1109/ICST.2013.20.

[57] A. Denisov and S. Pankevich. “Mull It Over: Mutation Testing Based on LLVM”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 25–31. DOI: 10.1109/ICSTW.2018.00024.

[58] Dorothy E Denning. “A lattice model of secure information flow”. In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[59] Anna Derezińska and Konrad Hałas. “Analysis of Mutation Operators for the Python Language”. In: *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*. Ed. by Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk. Cham: Springer International Publishing, 2014, pp. 155–164.

[60] Anna Derezinska and Karol Kowalski. “Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#”. In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. 2011, pp. 342–350.

[61] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. *Refactoring Test Code*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 2001.

[62] Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, and Lionel Briand. "Evolutionary robustness testing of data processing systems using models and data mutation (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 126–137.

[63] Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. "Augmenting field data for testing systems subject to incremental requirements changes". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26.1 (2017), pp. 1–40.

[64] Nicholas DiGiuseppe and James A. Jones. "On the Influence of Multiple Faults on Coverage-Based Fault Localization". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ' 11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 210–220. ISBN: 9781450305624. DOI: 10.1145/2001420.2001446. URL: https://doi.org/10.1145/2001420.2001446.

[65] Hyunsook Do and Gregg Rothermel. "On the use of mutation faults in empirical assessments of test case prioritization techniques". In: *IEEE Transactions on Software Engineering* 32.9 (2006), pp. 733–752.

[66] Avishek Sharma Dookhun and Leckraj Nagowah. "Assessing The Effectiveness Of Test-Driven Development and Behavior-Driven Development in an Industry Setting". In: *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. IEEE. 2019, pp. 365–370.

[67] V.H.S. Durelli, J. Offutt, and M.E. Delamaro. "Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing". In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. Apr. 2012, pp. 681–690. DOI: 10.1109/ICST.2012.158.

[68] Eduard Enoiu, Daniel Sundmark, Adnan Čaušević, and Paul Pettersson. "A comparative study of manual and automated testing for industrial control software". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 412–417.

[69] Anders Eriksson and Birgitta Lindström. "UML Associations: Reducing the gap in test coverage between model and code". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2016, pp. 589–599.

[70] Anders Eriksson, Birgitta Lindström, Sten F Andler, and Jeff Offutt. "Model transformation impact on test artifacts: An empirical study". In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. 2012, pp. 5–10.

[71] X. Feng, S. Marr, and T. O'Callaghan. "ESTP: An Experimental Software Testing Platform". In: *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*. 2008, pp. 59–63.

[72] Lloyd D Fosdick and Leon J Osterweil. "Data flow analysis in software reliability". In: *ACM Computing Surveys (CSUR)* 8.3 (1976), pp. 305–330.

[73] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. "All-uses vs mutation testing: an experimental comparison of effectiveness". In: *Journal of Systems and Software* 38.3 (1997), pp. 235–253.

[74] Gordon Fraser and Andrea Arcuri. "It is not the length that matters, it is how you control it". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE. 2011, pp. 150–159.

[75] Gordon Fraser and Andrea Arcuri. "Whole test suite generation". In: *IEEE Transactions on Software Engineering* 39.2 (2012), pp. 276–291.

[76] Gordon Fraser and Andreas Zeller. "Mutation-Driven Generation of Unit Tests and Oracles". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA ' 10. Trento, Italy: Association for Computing Machinery, 2010, pp. 147–158. ISBN: 9781605588230. DOI: 10.1145/1831708.1831728. URL: https://doi.org/10.1145/1831708.1831728.

[77] Gordon Fraser and Andreas Zeller. "Mutation-driven generation of unit tests and oracles". In: *IEEE Transactions on Software Engineering* 38.2 (2011), pp. 278–292.

[78] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. "Automated Oracle Data Selection Support". In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1119–1137.

[79] Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Milán Imre Gyalai. "Effect of test completeness and redundancy measurement on post release failures—An industrial experience report". In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10.

[80] Ralf Gerlich and Christian R Prause. "Evaluating test data generation for untyped data structures using genetic algorithms". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2018, pp. 126–129.

[81] Mechelle Gittens, Hanan Lutfiyya, Michael Bauer, David Godwin, Yong Woo Kim, and Pramod Gupta. "An empirical evaluation of system and regression testing". In: *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*. 2002, p. 3.

[82] M. Gligoric, V. Jagannath, and D. Marinov. "MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code". In: *2010 Third International Conference on Software Testing, Verification and Validation*. 2010, pp. 55–64.

[83] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. "Selective mutation testing for concurrent code". In: *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. 2013, pp. 224–234.

[84] P. Gong, R. Zhao, and Z. Li. "Faster mutation-based fault localization with a novel mutation execution strategy". In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107448.

[85] Google. *Google C++ Testing Framework*. May 2015.

[86] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. "On The Limits of Mutation Reduction Strategies". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. May 2016, pp. 511–522.

[87] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Mutations: How close are they to real faults?" In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE. 2014, pp. 189–200.

[88] R.G. Hamlet. "Testing Programs with the Aid of a Compiler". In: *Software Engineering, IEEE Transactions on* SE-3.4 (July 1977), pp. 279–290. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.231145.

[89] Li Hao, Jianqi Shi, Ting Su, and Yanhong Huang. "Automated Test Generation for IEC 61131-3 ST Programs via Dynamic Symbolic Execution". In: *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE. 2019, pp. 200–207.

[90] F. Hariri and A. Shi. "SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 860–863.

[91] Hadi Hemmati, Syed S Arefin, and Howard W Loewen. "Evaluating specification-level MC/DC criterion in model-based testing of safety critical systems". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2018, pp. 256–265.

[92] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. "Prioritizing manual test cases in traditional and rapid release environments". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[93] W. Högerle, F. Steimann, and M. Frenkel. "More Debugging in Parallel". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Nov. 2014, pp. 133–143. DOI: 10.1109/ISSRE.2014.29.

[94] Anders Hovmöller. *Mutmut: a Python mutation testing system*. https://hackernoon.com/mutmut-a-python-mutation-testing-system-9b9639356c78. 2016.

[95] W.E. Howden. "Weak Mutation Testing and Completeness of Test Sets". In: *Software Engineering, IEEE Transactions on* SE-8.4 (July 1982), pp. 371–379. ISSN: 0098-5589. DOI: 10.1109/TSE.1982.235571.

[96] Song Huang, Sen Yang, Zhanwei Hui, Yongming Yao, Lele Chen, Jialuo Liu, and Qiang Chen. "Runtime-environment testing method for android applications". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2019, pp. 534–535.

[97] Itregister. *Plextest*. http://www.itregister.com.au/products/plextest. 2007.

[98] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. "Code coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 955–963.

[99] Yue Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *Software Engineering, IEEE Transactions on* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[100] Yue Jia and M. Harman. “Constructing Subtle Faults Using Higher Order Mutation Testing”. In: *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. Sept. 2008, pp. 249–258. DOI: 10.1109/SCAM.2008.36.

[101] Yue Jia and M. Harman. “MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language”. In: *Practice and Research Techniques, 2008. TAIC PART ’08. Testing: Academic Industrial Conference*. Aug. 2008, pp. 94–98. DOI: 10.1109/TAIC-PART.2008.18.

[102] Wei Jin and Alessandro Orso. “Automated support for reproducing and debugging field failures”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.4 (2015), pp. 1–35.

[103] James A. Jones, James F. Bowring, and Mary Jean Harrold. “Debugging in Parallel”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. London, United Kingdom: Association for Computing Machinery, 2007, pp. 16–26. ISBN: 9781595937346. DOI: 10.1145/1273463.1273468. URL: https://doi.org/10.1145/1273463.1273468.

[104] James A. Jones and Mary Jean Harrold. “Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’05. Long Beach, CA, USA: Association for Computing Machinery, 2005, pp. 273–282. ISBN: 1581139934. DOI: 10.1145/1101908.1101949. URL: https://doi.org/10.1145/1101908.1101949.

[105] René Just, Michael D. Ernst, and Gordon Fraser. “Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 315–326. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610388. URL: http://doi.acm.org/10.1145/2610384.2610388.

[106] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. Nov. 2011, pp. 612–615.

[107] Sahitya Kakarla, Selina Momotaz, and Akbar Siami Namin. “An evaluation of mutation and data-flow testing: A meta-analysis”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 366–375.

[108] Mehdi Kessis, Yves Ledru, and Gérard Vandome. “Experiences in coverage testing of a Java middleware”. In: *Proceedings of the 5th international workshop on Software engineering and middleware*. 2005, pp. 39–45.

[109] Sunint Kaur Khalsa and Yvan Labiche. “An extension of category partition testing for highly constrained systems”. In: *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE. 2016, pp. 47–54.

[110] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. “Where should we fix this bug? a two-phase recommendation model”. In: *IEEE transactions on software Engineering* 39.11 (2013), pp. 1597–1610.

[111] Yong Woo Kim. “Efficient use of code coverage in large-scale software development”. In: *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research.* 2003, pp. 145–155.

[112] K. N. King and A. Jefferson Offutt. “A Fortran Language System for Mutation-based Software Testing”. In: *Softw. Pract. Exper.* 21.7 (June 1991), pp. 685–718. ISSN: 0038-0644. DOI: 10.1002/spe.4380210704. URL: http://dx.doi.org/10.1002/spe.4380210704.

[113] M. Kintis and N. Malevris. “Using Data Flow Patterns for Equivalent Mutant Detection”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops.* 2014, pp. 196–205.

[114] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. “Detecting Trivial Mutant Equivalences via Compiler Optimisations”. In: *IEEE Transactions on Software Engineering* 44.4 (2018), pp. 308–333.

[115] Marinos Kintis. “Effective methods to tackle the equivalent mutant problem when testing software with mutation”. PhD thesis. PhD thesis, Department of Informatics, Athens University of Economics and ⋯, 2016.

[116] Marinos Kintis and Nicos Malevris. “MEDIC: A static analysis framework for equivalent mutant identification”. In: *Information and Software Technology* 68 (2015), pp. 1–17. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.07.009. URL: http://www.sciencedirect.com/science/article/pii/S0950584915001329.

[117] Claus Klammer, Georg Buchgeher, and Albin Kern. “A retrospective of production and test code co-evolution in an industrial project”. In: *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST).* IEEE. 2018, pp. 16–20.

[118] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. “Practitioners’ Expectations on Automated Fault Localization”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 165–176. ISBN: 9781450343909. DOI: 10.1145/2931037.2931051. URL: https://doi.org/10.1145/2931037.2931051.

[119] Markus Kusano and Chao Wang. “CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications”. In: *Proc. ASE.* IEEE, 2013, pp. 722–725.

[120] J Lawrence, Steven Clarke, Margaret Burnett, and Gregg Rothermel. “How well do professional developers test with code coverage visualizations? an empirical study”. In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05).* IEEE. 2005, pp. 53–60.

[121] Tien-Duy B Le, Richard J Oentaryo, and David Lo. “Information retrieval and spectrum based bug localization: Better together”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 2015, pp. 579–590.

[122] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 609–615. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032354.

[123] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. "Mutation testing in practice using Ruby". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–6.

[124] Xia Li and Lingming Zhang. "Transforming Programs and Tests in Tandem for Fault Localization". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133916. URL: https://doi.org/10.1145/3133916.

[125] *List of information system failures*. https://www.ipa.go.jp/sec/system/system_fault.html. 2020.

[126] Yong Liu, Zheng Li, Ruilian Zhao, and Pei Gong. "An Optimal Mutation Execution Strategy for Cost Reduction of Mutation-Based Fault Localization". In: *Inf. Sci.* 422.C (Jan. 2018), pp. 572–596. ISSN: 0020-0255. DOI: 10.1016/j.ins.2017.09.006. URL: https://doi.org/10.1016/j.ins.2017.09.006.

[127] A. A. Lôbo de Oliveira, C. Gonçalves Camilo-Junior, E. Noronha de Andrade Freitas, and A. M. Rizzo Vincenzi. "FTMES: A Failed-Test-Oriented Mutant Execution Strategy for Mutation-Based Fault Localization". In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2018, pp. 155–165. DOI: 10.1109/ISSRE.2018.00026.

[128] Niels Lohmann. *Mutate++*. https://github.com/nlohmann/mutate_cpp. 2017.

[129] Philipp Luchscheider and Sebastian Siegl. "Test profiling for usage models by deriving metrics from component-dependency-models". In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2013, pp. 196–204.

[130] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. "MuJava: An Automated Class Mutation System: Research Articles". In: *Softw. Test. Verif. Reliab.* 15.2 (June 2005), pp. 97–133. ISSN: 0960-0833. DOI: 10.1002/stvr.v15:2. URL: http://dx.doi.org/10.1002/stvr.v15:2.

[131] L. Madeyski and N. Radyk. "Judy - a mutation testing tool for java". In: *IET Software* 4.1 (2010), pp. 32–42.

[132] Pratyusha Madiraju and Akbar Siami Namin. "Paraμ - A Partial and Higher-Order Mutation Tool with Concurrency Operators". In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. 2011, pp. 351–356.

[133] Claudio Magalhães, João Andrade, Lucas Perrusi, and Alexandre Mota. "Evaluating an automatic text-based test case selection using a non-instrumented code coverage analysis". In: *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*. 2017, pp. 1–9.

[134] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. “Dynamic analysis for diagnosing integration faults”. In: *IEEE Transactions on Software Engineering* 37.4 (2010), pp. 486–508.

[135] P.R. Mateo and M.P. Usaola. “Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases”. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. Sept. 2012, pp. 646–649. DOI: 10.1109/ICSM.2012.6405344.

[136] Aditya P Mathur and W Eric Wong. “An empirical comparison of data flow and mutation-based test adequacy criteria”. In: *Software Testing, Verification and Reliability* 4.1 (1994), pp. 9–31.

[137] Jason McDonald, Leesa Murray, Peter Lindsay, and Paul Strooper. “Module testing embedded software-an industrial pilot project”. In: *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*. IEEE. 2001, pp. 233–238.

[138] Joan C Miller and Clifford J Maloney. “Systematic mistake analysis of digital computer programs”. In: *Communications of the ACM* 6.2 (1963), pp. 58–63.

[139] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. “Efficient JavaScript Mutation Testing”. In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 74–83.

[140] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. “Guided Mutation Testing for JavaScript Web Applications”. In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 429–444.

[141] S. Moon, Y. Kim, M. Kim, and S. Yoo. “Ask the Mutants: Mutating Faulty Programs for Fault Localization”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 153–162. DOI: 10.1109/ICST.2014.28.

[142] Ivan Moore. *Jester*. http://jester.sourceforge.net/. 2001.

[143] Jakub Možucha and Bruno Rossi. “Is mutation testing ready to be adopted industry-wide?” In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2016, pp. 217–232.

[144] Keigo Naitou, Akito Tanikado, Shinsuke Matsumoto, Yoshiki Higo, Shinji Kusumoto, Hiroyuki Kirinuki, Toshiyuki Kurabayashi, and Haruto Tanno. “Toward Introducing Automated Program Repair Techniques to Industrial Software Development”. In: *Proceedings of the 26th Conference on Program Comprehension*. ICPC ’ 18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 332–335. ISBN: 9781450357142. DOI: 10.1145/3196321.3196358. URL: https://doi.org/10.1145/3196321.3196358.

[145] Hiroyuki Nakagawa, Toshinobu Hasegawa, Shori Matsui, and Tatsuhiro Tsuchiya. “Visualization of Specification Coverage: A Case Study of a Web Application Development in Industry”. In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2017, pp. 77–80.

[146] Hiroyuki Nakagawa, Shori Matsui, and Tatsuhiro Tsuchiya. “A visualization of specification coverage based on document similarity”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 136–138.

[147] James Newsome and Dawn Xiaodong Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.” In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4.

[148] Iulia Nica, Gerhard Jakob, Kathrin Juhart, and Franz Wotawa. “Results of a comparative study of code coverage tools in computer vision”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2017, pp. 36–37.

[149] Simon de Lang Nico Jansen and Alex van Assem. *Stryker*. https://stryker-mutator.io/stryker/. 2016.

[150] Simon de Lang Nico Jansen and Alex van Assem. *Stryker.NET*. https://stryker-mutator.io/stryker-net/. 2019.

[151] Kazuki Nishiura, Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. “Mutation Analysis for JavaScript Web Applications Testing”. In: vol. 2013. Jan. 2013.

[152] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. “Experience Report: How Effective Is Automated Program Repair for Industrial Software?” In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020.

[153] A Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. “An experimental evaluation of data flow and mutation testing”. In: *Software: Practice and Experience* 26.2 (1996), pp. 165–176.

[154] A. Offutt. “The Coupling Effect: Fact or Fiction”. In: *Proceedings of the ACM SIGSOFT ’ 89 Third Symposium on Software Testing, Analysis, and Verification*. TAV3. Key West, Florida, USA: Association for Computing Machinery, 1989, pp. 131–140. ISBN: 0897913426. DOI: 10.1145/75308.75324. URL: https://doi.org/10.1145/75308.75324.

[155] A. J. Offutt and K. N. King. “A Fortran 77 Interpreter for Mutation Analysis”. In: *Papers of the Symposium on Interpreters and Interpretive Techniques*. SIGPLAN ’ 87. St. Paul, Minnesota, USA: Association for Computing Machinery, 1987, pp. 177–188. ISBN: 0897912357. DOI: 10.1145/29650.29669. URL: https://doi.org/10.1145/29650.29669.

[156] A. Jefferson Offutt. “Investigations of the Software Testing Coupling Effect”. In: *ACM Trans. Softw. Eng. Methodol.* 1.1 (Jan. 1992), pp. 5–20. ISSN: 1049-331X. DOI: 10.1145/125489.125473. URL: http://doi.acm.org/10.1145/125489.125473.

[157] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. “An Experimental Evaluation of Selective Mutation”. In: *Proceedings of the 15th International Conference on Software Engineering*. ICSE ’93. Baltimore, Maryland, USA: IEEE Computer Society Press, 1993, pp. 100–107. ISBN: 0-89791-588-7. URL: http://dl.acm.org/citation.cfm?id=257572.257597.

[158] A. Jefferson Offutt and Ronald H. Untch. “Mutation Testing for the New Century”. In: ed. by W. Eric Wong. Norwell, MA, USA: Kluwer Academic Publishers, 2001. Chap. Mutation 2000: Uniting the Orthogonal, pp. 34–44. ISBN: 0-7923-7323-5. URL: http://dl.acm.org/citation.cfm?id=571305.571314.

[159] Andrew Jefferson Offutt et al. “Automatic test data generation”. PhD thesis. Georgia Institute of Technology, 1988.

[160] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. “HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java”. In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 165–170.

[161] Leon J Osterweil. “Data Flow Analysis as an Aid in Documentation, Assertion, Generation, Validation, and Error Detection; CU-CS-055-74”. In: (1974).

[162] M. Papadakis and N. Malevris. “Automatic Mutation Test Case Generation via Dynamic Symbolic Execution”. In: *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. Nov. 2010, pp. 121–130. DOI: 10.1109/ISSRE.2010.38.

[163] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. “Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 936–946. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818867.

[164] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. “Mutation testing advances: an analysis and survey”. In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[165] Mike Papadakis and Yves Le Traon. “Metallaxis-FL: mutation-based fault localization”. In: *Software Testing, Verification and Reliability* 25.5-7 (2015), pp. 605–628.

[166] Mike Papadakis and Nicos Malevris. “Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing”. In: *Software Quality Journal* 19.4 (Dec. 2011), pp. 691–723. ISSN: 0963-9314. DOI: 10.1007/s11219-011-9142-y. URL: http://dx.doi.org/10.1007/s11219-011-9142-y.

[167] Bartosz Kazimierz Papis, Konrad Grochowski, Kamil Subzda, and Kamil Sijko. “Experimental evaluation of test-driven development with interns working on a real industrial project”. In: *IEEE Transactions on Software Engineering* (2020).

[168] Ali Parsai, Alessandro Murgia, and Serge Demeyer. “LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems”. In: *Fundamentals of Software Engineering*. Ed. by Mehdi Dastani and Marjan Sirjani. Cham: Springer International Publishing, 2017, pp. 148–163. ISBN: 978-3-319-68972-2.

[169] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. "Evaluating and Improving Fault Localization". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. May 2017, pp. 609–620. DOI: 10.1109/ICSE.2017.62.

[170] Goran Petrović and Marko Ivanković. "State of mutation testing at google". In: *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 2018, pp. 163–171.

[171] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. "An industrial application of mutation testing: Lessons, challenges, and research directions". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2018, pp. 47–53.

[172] D. L. Phan, Y. Kim, and M. Kim. "MUSIC: Mutation Analysis Tool with High Configurability and Extensibility". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2018, pp. 40–46.

[173] Strategic Planning. "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology* (2002).

[174] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. "Darwin: An approach to debugging evolving programs". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21.3 (2012), pp. 1–29.

[175] Xiao Qu, Mithun Acharya, and Brian Robinson. "Configuration selection using code change impact analysis for regression testing". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 129–138.

[176] Krishnamoorthi Ramasamy and Sahaaya Arul Mary. "Incorporating varying requirement priorities and costs in test case prioritization for new and regression testing". In: *2008 International Conference on Computing, Communication and Networking*. IEEE. 2008, pp. 1–9.

[177] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. "An Empirical Study on the Application of Mutation Testing for a Safety-Critical Industrial Software System". In: *Proceedings of the Symposium on Applied Computing*. SAC ' 17. Marrakech, Morocco: Association for Computing Machinery, 2017, pp. 1401–1408. ISBN: 9781450344869. DOI: 10.1145/3019612.3019830. URL: https://doi.org/10.1145/3019612.3019830.

[178] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. "Isolating failure causes through test case generation". In: *Proceedings of the 2012 international symposium on software testing and analysis*. 2012, pp. 309–319.

[179] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[180] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. "Lightweight fault-localization using multiple coverage types". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 56–66.

[181] David Schuler and Andreas Zeller. “Javalanche: Efficient mutation testing for Java”. In: *ESEC/FSE ’09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering.* Amsterdam, Aug. 2009, pp. 297–298. ISBN: 9781605580012. DOI: 10.1145/1595696.1595750.

[182] Eric Schulte. *llvm-mutate – mutate LLVM IR.* https://eschulte.github.io/llvm-mutate/. 2013.

[183] Sebastian Siegl, Martin Russer, and Kai-Steffen Hielscher. “Partitioning the requirements of embedded systems by input/output dependency analysis for compositional creation of parallel test models”. In: *2015 Annual IEEE Systems Conference (SysCon) Proceedings.* IEEE. 2015, pp. 96–102.

[184] Jacob Slonim, Michael Bauer, and Jillian Ye. “Software reliability assurance in early development phases: a case study in an industrial setting”. In: *1996 IEEE Aerospace Applications Conference. Proceedings.* Vol. 4. IEEE. 1996, pp. 279–295.

[185] Matt Staats, Gregory Gay, and Mats PE Heimdahl. “Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing”. In: *2012 34th International Conference on Software Engineering (ICSE).* IEEE. 2012, pp. 870–880.

[186] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. “Programs, Tests, and Oracles: The Foundations of Testing Revisited”. In: *Proceedings of the 33rd International Conference on Software Engineering.* ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 391–400. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985847. URL: http://doi.acm.org/10.1145/1985793.1985847.

[187] Chengnian Sun and Siau-Cheng Khoo. “Mining succinct predicated bug signatures”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 2013, pp. 576–586.

[188] Youcheng Sun, Martin Brain, Daniel Kroening, Andrew Hawthorn, Thomas Wilson, Florian Schanda, Francisco Javier Guzman Jimenez, Simon Daniel, Chris Bryan, and Ian Broster. “Functional requirements-based automated testing for avionics”. In: *2017 22nd international conference on engineering of complex computer systems (ICECCS).* IEEE. 2017, pp. 170–173.

[189] Ramsay Taylor and John Derrick. “Smother: an MC/DC analysis tool for Erlang”. In: *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang.* 2015, pp. 13–18.

[190] Susumu Tokumoto and Hiroaki Yoshida. *Analytic method and analyzing apparatus.* US Patent App. 15/009,268. Aug. 2017.

[191] Taketo Tsunoda, Hironori Washizaki, Yosiaki Fukazawa, Sakae Inoue, Yoshiiku Hanai, and Masanobu Kanazawa. “Evaluating the work of experienced and inexperienced developers considering work difficulty in sotware development”. In: *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* IEEE. 2017, pp. 161–166.

[192] Reel Two. *Jumble.* http://jumble.sourceforge.net/. 2007.

[193] Roland H. Untch. “Mutation-Based Software Testing Using Program Schemata”. In: *Proceedings of the 30th Annual Southeast Regional Conference*. ACM-SE 30. Raleigh, North Carolina: Association for Computing Machinery, 1992, pp. 285–291. ISBN: 0897915062. DOI: 10.1145/503720.503749. URL: https://doi.org/10.1145/503720.503749.

[194] Roland H. Untch. “Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method”. AAI9703410. PhD thesis. Clemson, SC, USA, 1995. ISBN: 0-591-09880-6.

[195] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. “Mutation Analysis Using Mutant Schemata”. In: *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’93. Cambridge, Massachusetts, USA: ACM, 1993, pp. 139–148. ISBN: 0-89791-608-5. DOI: 10.1145/154183.154265. URL: http://doi.acm.org/10.1145/154183.154265.

[196] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. “How to Design a Program Repair Bot? Insights from the Repairnator Project”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. May 2018, pp. 95–104.

[197] M. P. Usaola and P. R. Mateo. “Mutation Testing Cost Reduction Techniques: A Survey”. In: *IEEE Software* 27.3 (2010), pp. 80–86.

[198] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. “Faster Mutation Analysis via Equivalence modulo States”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 295–306. ISBN: 9781450350761. DOI: 10.1145/3092703.3092714. URL: https://doi.org/10.1145/3092703.3092714.

[199] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. “An empirical study of android test generation tools in industrial cases”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 738–748.

[200] Mark Weiser. “Program slicing”. In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[201] Andreas Windisch, Stefan Wappler, and Joachim Wegener. “Applying particle swarm optimization to software testing”. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 2007, pp. 1121–1128.

[202] E. Wong, T. Wei, Y. Qi, and L. Zhao. “A Crosstab-based Statistical Method for Effective Fault Localization”. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. Apr. 2008, pp. 42–51. DOI: 10.1109/ICST.2008.65.

[203] W Eric Wong and Jenny Li. “An integrated solution for testing and analyzing Java applications in an industrial setting”. In: *12th Asia-Pacific Software Engineering Conference (APSEC’05)*. IEEE. 2005, 8–pp.

[204] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. “A Survey on Software Fault Localization”. In: *IEEE Transactions on Software Engineering* 42.8 (Aug. 2016), pp. 707–740. ISSN: 2326-3881. DOI: 10.1109/TSE.2016.2521368.

[205] W. Eric Wong and Aditya P. Mathur. “Reducing the Cost of Mutation Testing: An Empirical Study”. In: *J. Syst. Softw.* 31.3 (Dec. 1995), pp. 185–196. ISSN: 0164-1212. DOI: 10.1016/0164-1212(94)00098-0. URL: http://dx.doi.org/10.1016/0164-1212(94)00098-0.

[206] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. “Crashlocator: Locating crashing faults based on crash stacks”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 204–214.

[207] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[208] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. “Automated test input generation for android: Are we really there yet in an industrial case?” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 987–992.

[209] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. “Is Operator-based Mutant Selection Superior to Random Mutant Selection?” In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 435–444. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806863. URL: http://doi.acm.org/10.1145/1806799.1806863.

[210] Jian Zhou, Hongyu Zhang, and David Lo. “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports”. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 14–24.

[211] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. “An Empirical Study of Fault Localization Families and Their Combinations”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. ISSN: 2326-3881. DOI: 10.1109/TSE.2019.2892102.

[212] 独立行政法人情報処理推進機構（IPA） 技術本部 ソフトウェア高信頼化センター（SEC）. *ソフトウェア開発データ白書 2016-2017*. 2020.

[213] 徳本 晋 and 石井 康嗣. “ミューテーション解析における非利用アサーションの実証評価”. In: *ウィンターワークショップ 2018・イン・宮島 論文集*. Vol. 2018. 情報処理学会. Jan. 2018, pp. 20–21.

[214] 池田 翔, 中野 大扉, 亀井 靖高, 佐藤 亮介, 鵜林 尚靖, 吉武 浩, and 矢川 博文. “企業内ソースコードに対する自動バグ修正技術適用の試み”. In: *信学技報* 118.471 (Mar. 2019), pp. 193–198.

Appendix A

Detailed Proof that the Ratio of Computational Cost in k-th Order Split-stream Execution is $k+1$

In Section 4.3.3, we presented a technique to reduce execution costs by splitting the state at runtime, higher order split-stream execution (HOSSE), and gave a brief proof that the cost reduction ratio is approximately $k+1$. Here we illustrate the detail proof.

The following is the definition of each variable.

- k : the order of mutation
- n : the number of mutation locations
- λ : the number of instructions per mutation location
- p : the number of seeded faults per mutation location
- L_j : the j-th mutation location
- $c_{L_j,k}$: total computational cost of k-th order SSE started from mutation location L_j.

In our model, mutation locations are assumed to be uniformly distributed, so we can find that program under test has λn instructions. The higher order split stream execution (HOSSE) shares the execution of instructions up to each mutation location by branching the execution states. This mechanism allows k-th order mutations to reuse the execution states in $(k-1)$-th order mutations, i.e., the k-th order mutations costs only the branched execution stream. In Fig. 4.8, the branched execution streams that affect the cost of k-th order mutations are colored in red.

From the theoretical HOSSE model we can obtain the following recurrence relation.

$$
\begin{aligned}
c_{L_j,k} &= p \cdot c_{L_{j+1},k-1} + \cdots + p \cdot c_{L_{n-k+1},k-1} \\
&= p \cdot \sum_{i=j}^{n-k} c_{L_{i+1},k-1} && \text{(A.1)} \\
c_{L_j,0} &= \lambda(n-j+1) && \text{(A.2)}
\end{aligned}
$$

Now we prove the following relation by induction of the recurrence relation.

$$
c_{L_j,k} = \lambda \cdot p^k \binom{n-j+1}{k+1} \tag{A.3}
$$

Proof. Base case: $k = 0$.

$$c_{L_j,0} = \lambda(n-j+1) = \lambda \cdot p^0 \binom{n-j+1}{1}$$

Inductive hypothesis: Assume

$$c_{L_j,k'-1} = \lambda \cdot p^{k'-1} \binom{n-j+1}{k'}$$

is true for some $k' - 1 \geq 0$.

Inductive step:

$$\begin{aligned}
c_{L_j,k'} &= p \cdot \sum_{i=j}^{n-k'} c_{L_{i+1},k'-1} \\
&= \lambda \cdot p^{k'} \left\{ \binom{n-j}{k'} + \binom{n-j+1}{k'} + \cdots + \binom{k'+1}{k'} + \binom{k'}{k'} \right\} \\
&\quad (\because \text{Inductive hypothesis}) \\
&= \lambda \cdot p^{k'} \left\{ \left(\binom{n-j+1}{k'+1} - \binom{n-j}{k'+1} \right) + \left(\binom{n-j}{k'+1} - \binom{n-j-1}{k'+1} \right) + \cdots \right. \\
&\quad \left. + \left(\binom{k'+2}{k'+1} - \binom{k'+1}{k'+1} \right) + \binom{k'}{k'} \right\} \\
&\quad (\because \binom{n}{k} = \binom{n+1}{k+1} - \binom{n}{k+1}) \\
&= \lambda \cdot p^{k'} \binom{n-j+1}{k'+1}
\end{aligned}$$

which prove the case for $k = k'$. □

For comparison, we calculate the computational cost of the naïve method. Since the mutation locations of k-th order mutants are selected from k out of n, the total number of k-th order mutants is $\binom{n}{k} \cdot p^k$. The execution cost of each mutants in naïve method is λn that means the execution takes all instructions. Then we can obtain that the computational cost of naïve k-th order mutation is

$$c_{\text{naïve}} = \lambda n \cdot \binom{n}{k} \cdot p^k \tag{A.4}$$

The computational cost of k-th order SSE starting from L_1 is

$$c_{L_1,k} = \lambda \cdot p^k \binom{n}{k+1}. \tag{A.5}$$

The ratio of the computational cost of the proposed method to the naïve method is

$$\begin{aligned}
\frac{c_{\text{naïve}}}{c_{L_1,k}} &= \frac{\lambda n \cdot \binom{n}{k} \cdot p^k}{\lambda \cdot p^k \binom{n}{k+1}} \\
&= \frac{n \cdot \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1}}{\frac{n \cdot (n-1) \cdots (n-k+1) \cdot (n-k)}{(k+1) \cdot k \cdot (k-1) \cdots 1}} \\
&= \frac{n \cdot (k+1)}{n-k} \\
&= \frac{1}{1 - \frac{k}{n}} \cdot (k+1).
\end{aligned} \tag{A.6}$$

In general, the number of mutation locations n is sufficiently large with respect to k, so we can find the following relation.

$$\frac{c_{\text{naïve}}}{c_{L_1,k}} \simeq k+1 \tag{A.7}$$