VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

NGUYEN THU TRANG

Automated Localization and Repair for Variability Faults in Software Product Lines

DOCTOR OF PHILOSOPHY DISSERTATION

Major: Software Engineering

Hanoi - 2024

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

NGUYEN THU TRANG

Automated Localization and Repair for Variability Faults in Software Product Lines

DOCTOR OF PHILOSOPHY DISSERTATION

Major: Software Engineering

Supervisor: Dr. Vo Dinh Hieu

Co-Supervisor: Assoc. Prof. Dr. Ho Si Dam

Hanoi - 2024

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

NGUYEN THU TRANG

Automated Localization and Repair for Variability Faults in Software Product Lines

DOCTOR OF PHILOSOPHY DISSERTATION

Major: Software Engineering

Supervisor: Dr. Vo Dinh Hieu

Co-Supervisor: Assoc. Prof. Dr. Ho Si Dam

Hanoi - 2024

Acknowledgement

I am deeply grateful to the following individuals and organizations for their invaluable support and encouragement throughout the journey of completing my doctoral dissertation.

I would like to express my great appreciation to my supervisor, Dr. Vo Dinh Hieu, who is always willing to give me advice and comments on my problems. His constant support, guidance, and encouragement have been invaluable throughout the entire process. I feel very fortunate to be a student under the supervision of Dr. Vo Dinh Hieu.

I also would like to extend my sincere appreciation to my co-supervisor, Assoc. Prof. Ho Si Dam who gives me many valuable comments to improve my research and complete my dissertation.

I am grateful to Dr. Nguyen Van Son, who teaches me not only research skills but also presentation and writing skills. Without his expertise and encouragement, the completion of this dissertation would not have been possible.

I would like to thank MSc. Ngo Kien Tuan who is always willing to discuss with me and helps me a lot in conducting experiments.

My gratitude also extends to my teachers at the Department of Software Engineering, Assoc. Prof. Pham Ngoc Hung, Assoc. Prof. Dang Duc Hanh, Dr. Vu Thi Hong Nhan, my colleagues, and friends at UET-VNU. Without their knowledge and support, this dissertation would not have been successful.

I am thankful to Vingroup Innovation Foundation (VINIF) and The Development Foundation of Vietnam National University, Hanoi for providing financial support for this research. Their investment in my academic pursuits has been crucial in enabling the successful completion of this dissertation.

Lastly, I want to express my deepest gratitude to my family, who stand by me with unwavering support, patience, and understanding. Their encouragement, love, and belief in my abilities sustained me through the challenges of this doctoral journey.

Declaration

I hereby declare that this Doctoral Dissertation was carried out by me for the degree of Doctor of Philosophy under the guidance and supervision of my supervisors.

This dissertation is my own work and includes nothing, which is the outcome of work done in collaboration except as specified in the text.

It is not substantially the same as any I have submitted for a degree, diploma or other qualification at any other university; and no part has already been, or is currently being submitted for any degree, diploma or other qualification.

Hanoi, February 2024

Author

Nguyen Thu Trang

Abstract

Software Product Line (SPL) systems are becoming popular and widely employed to develop large industrial projects. However, their inherent variability characteristics pose extreme challenges for assuring the quality of these systems. Although automated debugging in single-system engineering has been studied in-depth, debugging SPL systems remains mostly unexplored. In practice, debugging activities in SPL systems are often performed manually in an ad-hoc manner. This dissertation sheds light on the automated debugging SPL systems by focusing on three fundamental tasks, including *false-passing* product detection, variability fault localization, and variability fault repair.

First, *this dissertation aims to improve the reliability of the test results by detecting false-passing products in SPL systems failed by variability bugs.* Given a set of tested products of an SPL system, the proposed approach, CLAP, collects failure indications in failing products based on their implementation and test quality. For a passing product, CLAP evaluates these indications, and the stronger the indications, the more likely the product is *false-passing*. Specifically, the possibility of the product being *false-passing* is evaluated based on if it has a large number of statements that are highly suspicious in the failing products and if its test suite is lower quality compared to the failing products' test suites.

Second, *this dissertation presents* VARCOP*, a novel and effective variability fault localization approach.* For an SPL system failed by variability bugs, VARCOP isolates suspicious code statements by analyzing the overall test results of the sampled products and their source code. The isolated suspicious statements are the statements related to the interaction among the features that are necessary for the visibility of the bugs in the system. In VARCOP, the suspiciousness of each isolated statement is assessed based on both the overall test results of the products containing the statement as well as the detailed results of the test cases executed by the statement in these products.

Third, *this dissertation proposes two approaches, product-based and system-based, to repair the variability bugs in an SPL system to fix the failures of the failing products and not to break the correct behaviors of the passing products.* For the product-based approach, each failing product is fixed individually, and the obtained patches are then propagated and validated on the other products of the system. For the system-based approach, all the products are repaired simultaneously. The patches are generated and validated by all the sampled products of the system in each repair iteration. Moreover, to improve the repair performance of both approaches, this dissertation also introduces several heuristic rules for effectively and efficiently deciding where to fix (*navigating modification points*) and how to fix (*selecting suitable modifications*). These heuristic rules use intermediate validation results of the repaired programs as feedback to refine the fault localization results and

evaluate the suitability of the modifications before actually applying and validating them by test execution.

To evaluate the proposed approaches, this dissertation conducted several experiments on a large public dataset of buggy SPL systems. The experimental results show that CLAP can effectively detect *false-passing* and *true-passing* products with an average accuracy of more than 90%. Especially, the precision of *false-passing* product detection by CLAP is up to 96%. This means among ten products predicted as *false-passing* products, more than nine products are precisely detected.

For variability fault localization, VARCOP significantly improves two state-of-the-art techniques by 33% and 50% in ranking the incorrect statements in the systems containing a single bug each. In about two-thirds of the cases, VARCOP correctly ranks the buggy statements at the top-3 positions in the ranked lists. For the cases containing multiple bugs, VARCOP outperforms the state-of-the-art approaches two times and ten times in the proportion of bugs localized at the top-1 positions.

Furthermore, for repairing variability faults, the experimental results show that the product-based approach is around 20 times better than the system-based approach in the number of correct fixes. Notably, the heuristic rules could improve the performance of both approaches by increasing of 30-150% the number of correct fixes and decreasing of 30-50% the number of attempted modification operations.

Keywords: *Software product line*, *variability fault*, *coincidential correctness*, *fault localization*, *automated program repair*

Contents

Acknowledgement

Declaration **i**

Abstract **ii**

TABLE OF CONTENTS **iv**

List of Figures **viii**

List of Tables **x**

Acronyms **xii**

Chapter 1 Introduction **1**

1.1 Problem Statement 1

1.2 Objective and Contributions 6

1.3 Dissertation Outline 11

Chapter 2 Background and Literature Review **12**

2.1 Background 12

2.1.1 Software Product Line 12

2.1.2 Testing Software Product Lines 17

2.1.3 Fault Localization 20

2.1.4 Automated Program Repair 22

2.2 Literature Review 25

2.3 Benchmarks for Software Product Lines 28

Chapter 3 False-passing Product Detection **31**

3.1 Introduction 31

3.2 Motivation and Problem Formulation 33

3.2.1 Motivation 33

3.2.2 Problem Formulation 34
3.3 *False-passing* Product Detection 36
3.3.1 Suspiciousness of Product Implementation 38
3.3.2 Test Adequacy 41
3.3.3 Test Effectiveness 44
3.3.4 Detecting *False-passing* Products 48
3.4 Mitigation of Negative Impact of *False-passing* Products on Variability Fault Localization 49
3.5 Empirical Methodology 50
3.5.1 Research Questions 50
3.5.2 Dataset 51
3.5.3 Empirical Procedure 53
3.5.4 Metrics 55
3.5.5 Experimental Setup 56
3.6 Experimental Results 57
3.6.1 Accuracy Analysis (RQ1) 57
3.6.2 Mitigating Impact of *False-passing* Products on Fault Localization (RQ2) 58
3.6.3 Sensitivity Analysis (RQ3) 61
3.6.4 Intrinsic Analysis (RQ4) 64
3.6.5 Time Complexity (RQ5) 66
3.6.6 Threats to Validity 66
3.7 Summary 67

Chapter 4 Variability Fault Localization **69**
4.1 Introduction 69
4.2 Motivating Example 71
4.2.1 An Example of Variability Faults in Software Product Lines 71
4.2.2 Observations 73
4.2.3 VarCop Overview 75
4.3 Feature Interaction 76
4.3.1 Feature Interaction Formulation 77
4.3.2 The Root Cause of Variability Failures 78
4.4 Buggy Partial Configuration Detection 80

4.4.1 Buggy Partial Configuration 81
4.4.2 Important Properties to Detect Buggy Partial Configuration 83
4.4.3 Buggy Partial Configuration Detection Algorithm 86
4.5 Suspicious Statement Identification 86
4.6 Suspicious Statement Ranking 88
4.6.1 Product-based Suspiciousness Assessment 88
4.6.2 Test Case-based Suspiciousness Assessment 89
4.6.3 Assessment Combination 90
4.7 Empirical Methodology 90
4.7.1 Dataset 91
4.7.2 Evaluation Setup, Procedure, and Metrics 92
4.8 Empirical Results 93
4.8.1 Accuracy and Comparison (RQ1) 93
4.8.2 Intrinsic Analysis (RQ2) 99
4.8.3 Sensitivity Analysis (RQ3) 103
4.8.4 Performance in Localizing Multiple Bugs (RQ4) 105
4.8.5 Time Complexity (RQ5) 107
4.8.6 Threats to Validity 108
4.9 Summary 109

Chapter 5 Automated Variability Fault Repair **110**
5.1 Introduction 110
5.2 Problem Statement 113
5.3 Automated Variability Fault Repair 115
5.3.1 Product-based Approach ($ProdBased_{basic}$) 116
5.3.2 System-based Approach ($SysBased_{basic}$) 120
5.3.3 Product-based Approach vs System-based Approach 122
5.4 Heuristic Rules for Improving the Repair Performance 123
5.4.1 Heuristic Rules for Improving the Performance of Automated Program Repair Tools 123
5.4.2 Applying the Heuristic Rules in Repairing Variability Faults 128
5.5 Experiment Methodology 130
5.5.1 Benchmarks 132
5.5.2 Evaluation Procedure and Metrics 133

5.6 Experimental Results 136
5.6.1 RQ1. Performance Analysis 136
5.6.2 RQ2. Intrinsic Analysis 141
5.6.3 RQ3. Sensitivity Analysis 146
5.6.4 Threats to Validity 150
5.7 Summary 151

Chapter 6 Conclusion **153**

List of Publications **157**

References **158**

List of Figures

1.1 The proposed debugging process of SPL systems 2

2.1 Overview of an engineering process for software product lines[1] 13
2.2 An example of feature model of Elevator system 14
2.3 SPL testing interest: actual test of products [2] 17
2.4 Example of sampling algorithms [3] 18
2.5 Program spectrum of a program with n elements and m test cases 20
2.6 Example of program spectrum and FL results by Tarantula and Ochiai . . . 22
2.7 Standard steps in the pipeline of the test-suite-based program repair 22

3.1 CLAP's overview 38
3.2 The presence of the suspicious statements in the passing products 39
3.3 The presence of bug-involving statements in the passing products 41
3.4 The portion of suspicious statements in the passing products which are not covered by their test suites 42
3.5 The undiagnosability (DDU') of the passing products' test suites 43
3.6 The incorrectness verification of the passing products' test suites 46
3.7 The correctness reflectability of the passing products' test suites 48

4.1 VARCOP's Overview 75
4.2 *Hit@1–Hit@5* of VARCOP, S-SBFL and SBFL 96
4.3 Performance by number of involving features of bugs 99
4.4 Impact of *Buggy PC* Detection on performance 100
4.5 Impact of Normalization on performance 101
4.6 Impact of choosing $score(s, M)$ on performance 102
4.7 Impact of choosing combination weight on performance 103
4.8 Impact of the sample size on performance 104
4.9 Impact of the size of test set on performance 105
4.10 VARCOP, S-SBFL and SBFL in localizing multiple bugs 106

5.1 The feature model of the ExamDB system . . . 113
5.2 The process of APR with the two proposed heuristic rules . . . 128
5.3 RQ2 – Impact of the suitability threshold θ on $ProdBased_{enhanced}$'s performance . . . 144
5.4 RQ2 – Impact of the suitability parameters (α, β) on $ProdBased_{enhanced}$'s performance . . . 145
5.5 RQ3 – The performance of $ProdBased_{enhanced}$ in fixing variability bugs of different SPL systems . . . 147
5.6 RQ3 – Impact of the number of failing products on $ProdBased_{enhanced}$'s performance – BankAccount . . . 148
5.7 RQ3 – Impact of the number of suspicious statements on $ProdBased_{enhanced}$'s performance – BankAccount . . . 150

List of Tables

2.1 The sampled products and their overall test results 15
2.2 Several popular SBFL formulae [4] 21
2.3 Dataset Statistics [5] 30

3.1 Empirical study about the impact of *false-passing* products on variability fault localization performance (in *Rank*) 35
3.2 Products' test suites before and after being transformed 51
3.3 Dataset overview 53
3.4 Accuracy of *false-passing* product detection model 57
3.5 Mitigating the *false-passing* products' negative impact on FL performance . 58
3.6 Impact of different experimental scenarios 61
3.7 Clap's performance on each system in system-based edition 62
3.8 Clap's performance on each system in within-system edition 62
3.9 Impact of different training data sizes (the number of systems) 64
3.10 Impact of attributes on Clap's performance 65

4.1 The sampled products and their overall test results 72
4.2 Dataset Statistics [5] 91
4.3 Performance of VarCop, SBFL, the combination of Slicing and SBFL (*S-SBFL*), and Arrieta et al. [6] (*FB*) 94
4.4 Performance by Mutation Operators 97
4.5 Performance by Code Elements of Bugs 98

5.1 The tested products of ExamDB system and their test results 114
5.2 Example of modification operations for fixing the bug at statement s_5 in Listing 5.1 124
5.3 Benchmarks 132

5.4 RQ1 – The performance of repairing variability bugs of the approaches in the setting *withoutFL* (i.e., the correct positions of buggy statements are given) . . . 136
5.5 RQ1 – The performance of repairing variability bugs of the approaches in the setting *withFL* . . . 138
5.6 RQ1 – Statistical analysis regarding #Correct fixes of $ProdBased_{enhanced}$ vs $ProdBased_{basic}$ and $SysBased_{enhanced}$ vs $SysBased_{basic}$ in different experiment executions – *withFL* setting . . . 140
5.7 RQ2 – Impact of disabling each heuristic rule in $ProdBased_{enhanced}$ . . . 141
5.8 RQ2 – Impact of the similarity functions in modification suitability measurement . . . 142

Acronyms

Buggy PC Buggy Partial Configuration

APR Automated Program Repair

DDU Density-Diversity-Uniqueness

FL Fault Localization

KNN K-Nearest Neighbor

LSTM Long Short-Term Memory

QA Quality Assurance

SBFL Spectrum-Based Fault Localization

SMT Satisfiability Modulo Theories

SPL Software Product Line

SVM Support Vector Machine

Chapter 1

Introduction

1.1 Problem Statement

Nowadays, Software Product Line (SPL) systems (or Configurable Systems, in general) are becoming popular and widely employed to develop large industrial projects [7–9]. SPL engineering creates remarkable efficiencies in developing software products. Instead of developing each software product from scratch, SPL methodology allows one to easily and quickly construct multiple products from reusable artifacts. This helps to improve productivity, increase market agility, and reduce development costs. Companies and institutions such as NASA, Hewlett Packard, General Motors, Boeing, Nokia, and Philips apply SPL technology with great success to broaden their software portfolio [10].

An SPL system is a product family containing a set of products sharing a common code base. Each product is identified by the selected features [7]. In other words, a project adopting the SPL methodology can tailor its functional and nonfunctional properties to the requirements of users [7, 11]. This has been done using a very large number of *options* which are used to control different *features* [11] additional to the *core software*. A set of *selections* of all the features (*configurations*) defines a program *variant* (*product*). For example, Linux Kernel supports thousands of features controlled by +12K compile-time options that can be configured to generate specific kernel *variants* for billions of scenarios.

Another popular example of an SPL system is WordPress, a powerful tool for building websites. WordPress allows users to easily customize their own websites by providing a lot of features implemented as plugins. By 60K plugins [1], multiple variants of websites can be created, from simple websites such as personal blogs, photo blogs, or business websites to complex ones like enterprise applications.

Although the variability of SPL system creates many benefits in software developments, this charateristic challenges Quality Assurance (QA) [3, 12–15]. In comparison with the traditional single-system engineering (aka. non-configurable system), fault detection, localization, and repair through testing in SPL systems are more problematic, as a bug can

[1] https://wordpress.org/plugins/

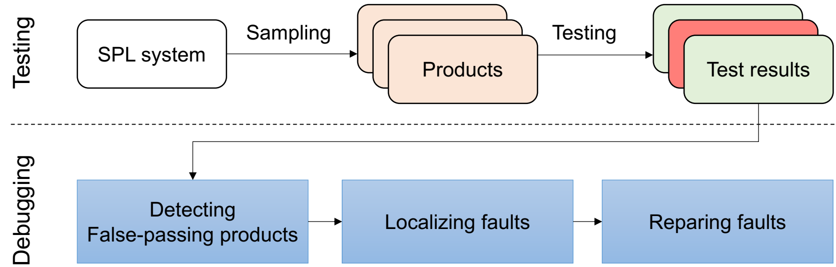


Figure 1.1: The proposed debugging process of SPL systems

be *variable* (so-called *variability bug*), which can only be exposed under *certain* combinations of the system features [12, 16]. In particular, there exists a set of features that must be selected to be *on* and *off together* to necessarily reveal the bug. Due to the presence/absence of the *interaction* among the features in such set, the buggy statements behave differently in the products where these features are on and off together or not. Hence, *the incorrect statements can only expose their bugginess in certain products, yet cannot in others.* Specially in an SPL system, variability bugs only cause failures in certain products, while the others still pass all their tests.

In general, to guarantee the quality of a system during development and before release, developers need to detect and address software faults. In practice, testing is one of the most popular and practical techniques employed to determine whether the program exhibits as expected. If a fault is detected, e.g., a test failed, developers need to localize and repair it. This debugging process can be done manually or automatically. Several techniques have been introduced for automated debugging a single-system, such as Tarantula [17] for localizing faults and GenProg [18] for repairing faults.

To guarantee the quality of an SPL system, a family of software products, the similar QA process is also adopted [15]. Specifically, for detecting bugs in an SPL system, each product/variant of the system is constructed and tested against the designed test suite. However, due to the exponential growth of possible configurations, a subset of products are systematically selected by sampling techniques such as *t-wise* [19], *statement-coverage* [20], or *one-disabled* [14]. After that, each sampled product is validated against its test suite. If the system contains variability bugs, such bugs could cause several products to fail their tests (*failing products*), and the others still pass all their tests (*passing products*).

After the faults are detected (i.e., failed tests), the debugging process includes two main tasks: *fault localization* and *fault repair*. In practice, testing results are often leveraged

by Fault Localization (FL) approaches to pinpoint the position of the bugs and used to evaluate the correctness of *patches* generated by Automated Program Repair (APR) tools. However, the *unreliability* of the test results (i.e., *coincidental correctness*) could negatively impact the performance of the debugging tools [21]. Thus, the buggy products which coincidentally passed all their tests (*false-passing* products) must be detected and eliminated before leveraging the test results for localizing and repairing faults.

Although automated debugging in single-system engineering has been studied in-depth, debugging SPL systems still remains mostly unexplored. This dissertation focuses on automated debugging SPL systems in three main tasks, including detecting *false-passing* products, localizing variability faults, and repairing such faults in SPL systems. The proposed process for automated debugging SPL system is shown in the bottom half of Figure 1.1. Due to the dynamic nature of SPL systems, with numerous combinations and interactions among features, it amplifies the difficulties of debugging SPL systems. The subsequent paragraphs introduce the details of each problem focused in this dissertation.

***False-passing* product detection**. Thorough testing is often required to guarantee the quality of software program. However, it is often hard, tedious, and time-consuming to conduct thorough testing in practice. Various bugs could be neglected by the test suites since it is extremely difficult to cover all the programs' behaviors. Moreover, there are kinds of bugs which are challenging to be detected due to their difficulties in infecting the program states and propagating their incorrectness to the outputs [22]. Consequently, even when they reached the defects, there are test cases that still obtain correct outputs. Such test cases are called *coincidentally correct/passed* tests. Indeed, coincidental correctness is a prevalent problem in software testing [21], and this phenomenon causes a severely negative impact on fault localization performance [21, 23, 24].

Similar to testing for non-configurable code, the coincidental correctness phenomenon also happens in SPL systems and causes difficulties in finding faults in these systems. For a buggy SPL system, the bugs could be in one or more products. Ideally, if a product contains bugs (*buggy products*), the bugs should be revealed by its test suite, i.e., there should be at least a failed test after testing. However, if the test suite of the product is ineffective in detecting the bugs, the product's overall test result would be passing. For instance, the test suite does not cover the product's buggy statements or those test cases could reach the buggy statements but could not propagate the incorrectness to the outputs, the product still passes all the tests. Concequently, a passing product is indeed

a buggy product, yet incorrectly considered as passing. That passing product is namely a *false-passing* product.

Due to their unreliability of the test results, these *false-passing* products might negatively impact the fault localization performance. In particular, the performance of two main spectrum-based FL strategies, *product-based* and *test case-based*, is directly affected.

First, the *product-based* fault localization techniques [6] evaluate the suspiciousness of a statement in a buggy SPL system based on the appearance of the statement in failing and/or passing products. Specially, the key idea to find bugs in an SPL system is that a statement which is included in more failing products and fewer passing products is more likely to be buggy than the other statements of the system. Misleadingly counting a buggy product as a passing product incorrectly decreases the number of failing products and increases the number of passing products containing the buggy statement. Consequently, the buggy statement is considered less suspicious than it should be.

Second, the *test case-based* fault localization techniques [25] measure the suspicious scores of the statements based on the numbers of failed and/or passed tests executed by them. Indeed, *false-passing* products could lead to under-counting the number of failed tests and over-counting the number of passed tests executed by the buggy statements. The reason is that *false-passing* products contain bugs, but there is no failed test. In these *false-passing* products, the buggy statements are not executed by any test, or they are reached by several tests, yet those tests *coincidentally* passed. Both low coverage test suite and coincidentally passed tests can cause inaccurate evaluation for the statements.

Variability fault localization. Despite the importance of *variability fault localization*, the existing fault localization approaches [4, 6, 25] are not designed for this kind of bugs. These techniques are specialized for finding bugs in a particular product. For instance, to isolate the bugs causing failures in multiple products of a single SPL system, the slice-based methods [25–27] could be used to identify all the failure-related slices for each product independently of others. Consequently, there are multiple sets of (large numbers of) isolated statements that need to be examined to find the bugs. This makes the slice-based methods [25] become impractical in SPL systems.

In addition, the state-of-the-art technique, Spectrum-Based Fault Localization (SBFL) [4, 28–31] can be used to calculate the suspiciousness scores of code statements based on the test information (i.e., program spectra) of each product of the system separately. For each product, it produces a ranked list of suspicious statements. As a result, there might be

multiple ranked lists produced for a single buggy SPL system. From these multiple lists, developers cannot determine a starting point to diagnose the root causes of the failures. Hence, it is inefficient to find variability bugs by using SBFL to rank suspicious statements in multiple variants separately.

Another method to apply SBFL for localizing variability bugs in an SPL system is that one can treat the whole system as a single program [5]. This means that the mechanism controlling the presence/absence of the features in the system (e.g., the preprocessor directives `#ifdef`) would be considered as the conditional `if-then` statements during the FL process. Note that, this dissertation considers the product-based testing [32, 33]. Specially, each product is tested individually with its own test set. Additionally, a test, which is designed to test a feature in domain engineering, is concretized to multiple test cases according to products' requirements in application engineering [32]. The suspiciousness score of a statement is measured based on the total numbers of the passed and failed tests executed by it in all the tested products. By this adaptation of SBFL, a single ranked list of the statements for a buggy SPL system can be produced according to the suspiciousness score of each statement. Meanwhile, the characteristics including the interactions between system features and the variability of failures among products are also useful to isolate and localize variability bugs in SPL systems. However, these kinds of important information are not utilized in the existing approaches.

Automated variability fault repair. After localizing faults, developers still need to spend a large amount of their time on fixing them [34]. Moreover, with the variability characteristics of SPL systems, addressing bugs in SPL systems could be much more complicated. Echeverría et al. [35] conducted an empirical study to evaluate engineers' behaviors in fixing errors and propagating the fixes to other products in an industrial SPL system. They showed that fixing SPL systems is very challenging, especially for large systems. Indeed, in an SPL system, each product is composed of a different set of features. Due to the interaction of different features, a *variability bug* in an SPL system could manifest itself in some products of the system but not in others. To fix variability bugs, APR approaches need to find patches which not only work for a single product but also for all the products of the system. In other words, APR approaches need to fix the incorrect behaviors of all *failing products*, and do not break the correct behaviors of the *passing products*.

To reduce the cost of software maintenance and alleviate the heavy burden of manu-

ally debugging activities, multiple automated program repair approaches [18, 36–40] have been proposed in recent decades. These approaches employ different techniques to automatically (i.e., without human intervention) synthesize patches that eliminate program faults and obtain promising results. However, these approaches focus on fixing bugs in a single non-configurable system. These approaches cannot be directly applied for fixing incorrect code statements in SPL systems since they only fix a single product individually without considering the mutual behaviors among the shared features of the products. Consequently, the generated patches could be fit for only the product under repair, yet could not work for the whole SPL system.

In the context of SPL systems, there are several studies attempting to deal with the variability bugs at different levels, such as model or configuration. For example, Arcaini et al. [41, 42] attempt to fix bugs in the variability models. Weiss et al. [43, 44] repair misconfigurations of the SPL systems. However, automated repair variability bugs at the source code level still needs further investigation.

In summary, SPL systems are widely adopted in industry. A variability bug of the SPL system could cause severe damage since it could be included in and cause failures for multiple products of the system. In addition, the inherent variability characteristics of SPL systems pose extreme challenges for detecting, localizing, and fixing variability bugs. This dissertation sheds light on the automated debugging buggy SPL systems by focusing on three fundamental tasks, including *false-passing* product detection, variability fault localization, and variability fault repair.

1.2 Objective and Contributions

This dissertation aims to propose approaches for automatically debugging SPL systems failed by variability bugs. To improve the reliability of the test results, this dissertation proposes CLAP, an approach for detecting *false-passing* products. Next, this dissertation presents VARCOP, a novel FL approach specialized for variability faults of SPL systems. Finally, this dissertation introduces two *product-based* and *system-based* approaches to automatically repairing variability faults.

First, *this dissertation introduces* CLAP*, an approach for* ***detecting false-passing products*** *of buggy SPL systems.* The intuition of the proposed approach is that for a buggy SPL system, the sampled products can share some common functionalities. If the unex-

pected behaviors of the functionalities are revealed by the tests in some (failing) products, the other products having similar functionalities are likely to be caused failures by those unexpected behaviors. In CLAP, *false-passing* products can be detected based on the *failure indications* which are collected by reviewing the *implementation* and *test quality* of the failing products. To evaluate the possibility that a passing product is a *false-passing* one, CLAP proposes several *measurable attributes* to assess the strength of these failure indications in the product. The stronger indications, the more likely the product is *false-passing*.

The proposed attributes are belonged to two aspects: *product implementation* (products' source code) and *test quality* (the adequacy and the effectiveness of test suites). The attributes regarding *product implementation* reflect the possibility that the passing product contains bugs. Intuitively, if the product has more (suspicious) statements executing the tests failed in the failing products of the system, the product is more likely to contain bugs. For the *test quality* of the product, the *test adequacy* reflects how its suite covers the product's code elements such as statements, branches, or paths [45]. A low-coverage test suite could be unable to cover the incorrect elements in the buggy product. Hence, the product with a lower-coverage test suite is more likely to be *false-passing*. Meanwhile, the *test effectiveness* reflects how intensively the test suite verifies the product's behaviors and its ability to explore the product's (in)correctness [46, 47]. The intuition is that if the product is checked by a test suite which is less effective, its overall test result is less reliable. Then, the product is more likely to be a *false-passing* one.

Furthermore, *this dissertation discusses strategies to mitigate the impact of false-passing products on FL results*. Since the negative impact is mainly caused by the unreliability of the test results, this dissertation aims to improve the *reliability* of the test results by enhancing the test quality based on the failure indications. In addition, the reliability of test results could also be improved by *disregarding* the unreliable test results at either product-level or test case-level.

Second, *this dissertation proposes* VARCOP*, a novel approach for* ***localizing variability bugs***. The key ideas of VARCOP is that variability bugs are localized based on (i) the interaction among the features which are necessary to reveal the bugs, and (ii) the bugginess exposure which is reflected via both overall test results at the product-level and the detailed test results at the test case-level.

Particularly, for a buggy SPL system, VARCOP detects sets of the features which need to

be selected on/off together to make the system fail by analyzing the overall test results (i.e., the state of passing all tests or failing at least one test) of the products. This dissertation calls each of these sets of the feature selections a *Buggy **P**artial **C**onfiguration (Buggy PC)*. Then, VarCop analyzes the interaction among the features in these *Buggy PCs* to isolate the statements which are suspicious.

In VarCop, the suspiciousness of each isolated statement is assessed based on two criteria. The first criterion is based on the overall test results of the products containing the statement. By this criterion, the more failing products and the fewer passing products where the statement appears, the more suspicious the statement is. Meanwhile, the second one is assessed based on the suspiciousness of the statement in the failing products which contain it. Specially, in each failing product, the statement's suspiciousness is measured based on the detailed results of the products' test cases. The idea is that if the statement is more suspicious in the failing products based on their detailed test results, the statement is also more likely to be buggy in the whole system.

Third, *this dissertation proposes two approaches, product-based and system-based, for **automatically repairing variability faults** of the SPL systems.* For the *product-based approach* ($ProdBased_{basic}$), each failing product of the system is repaired individually, and then the obtained patches, which cause the product under repair to pass all its tests, are propagated and validated on the other products of the system. For the *system-based approach* ($SysBased_{basic}$), instead of repairing one individual product at a time, all the products are considered for repairing simultaneously. Specifically, the patches are generated and then validated by all the sampled products of the system in each repair iteration. For both approaches, the valid patches are the patches causing all the available tests of all the sampled products of the system to pass.

Furthermore, *this dissertation introduces several heuristic rules for improving the performance of the two approaches in repairing buggy SPL systems.* These heuristic rules are started from the observation that, in order to effectively and efficiently fix a bug, an APR tool must correctly decide (i) where to fix (*navigating modification points*) and (ii) how to fix (*selecting suitable modifications*). The heuristic rules focus on enhancing the accuracy of these tasks by leveraging intermediate validation results of the repair process.

For *navigating modification points*, APR tools [38, 48] often utilize the *suspiciousness scores*, which refer to the probability of the code elements to be faulty. These scores are often calculated once for all before the repair process by FL techniques such as

SBFL [25, 31]. However, a lot of additional information can be obtained during the repairing process, such as the modified programs' validation results. Such information can provide valuable feedback for continuously refining the navigation of the modification points [49]. Therefore, in this work, besides suspiciousness scores, the *fixing scores* of the modification points, which refer to the ability to fix the program by modifying the source code of the corresponding points, are used for navigating modification points in each repair iteration. The fixing scores are continuously measured and updated according to the intermediate validation results of the modified programs. The intuition is that *if modifying the source code at a modification point mp causes (some of) the initial failed test(s) to be passed, mp could be the correct position of the fault or have relations with the fault.* Otherwise, modifying its source code cannot change the results of the failed tests. The modification point with a high fixing score and high suspiciousness score should be prioritized to attempt in each subsequent repair iteration.

After a modification point is selected, APR tools generate and *select suitable modifications* for that point and evaluate them by executing tests [36, 38, 50]. This dynamic validation is time-consuming and costs a large amount of resources. In order to mitigate the wasted time of validating incorrect modifications, this dissertation introduces *modification suitability measurement* for lightweight evaluating and quickly eliminating unsuitable modifications. The suitability of a modification at position *mp* is evaluated by the similarity of that modification with the original source code and with the previous attempted modifications at *mp*. The intuition is that *the correct modification at mp is often similar to its original code and the other successful modifications at this point, while the modifications similar to the failed modifications are often incorrect.* Thus, the more similar a modification is to the original code and to the successful modifications, and the less similar it is to the failed modifications, then the more suitable that modification is for attempting at *mp*.

These heuristic rules are embedded on the product-based and system-based approaches, and the enhanced versions are called $ProdBased_{enhanced}$ and $SysBased_{enhanced}$.

The **research methodology** of the dissertation is the combination of qualitative research and quantitative research:

- Qualitative research includes: (i) Analyzing the concepts, ideas, methodologies, and techniques from prior studies; (ii) identifying strengths, weaknesses, and challenges of these approaches; (iii) enhancing, integrating, and proposing novel solutions for

addressing the problems.

- Quantitative research includes: (i) Investigating available datasets, (ii) conducting experiments, (iii) validating the effectiveness of proposed approaches, and (iv) publishing research findings for peer validation within the academic community.

Scope of the Dissertation: The dissertation focuses on addressing the problem of automated debugging buggy SPL systems, which contain variability bugs. Specifically, this dissertation focuses on three tasks, including *false-passing* product detection, variability fault localization, and variability fault repair.

In summary, this dissertation makes the following main contributions:

- The formulation of the *false-passing* product detection problem in SPL systems and a large benchmark for evaluating *false-passing* product detection techniques.
- CLAP: an effective approach to detect *false-passing* products in SPL systems and mitigate their negative impact on variability fault localization performance. CLAP's implementation can be found at: *https://ttrangnguyen.github.io/CLAP/*.
- A formulation of *Buggy Partial Configuration (Buggy PC)* where the interaction among the features in the *Buggy PC* is the root cause of the failures caused by variability bugs in SPL systems.
- VARCOP: A novel effective approach/tool to localize variability bugs in SPL systems. VARCOP's implementation can be found at: *https://ttrangnguyen.github.io/VARCOP/*.
- Heuristic rules for navigating modification points and selecting suitable modifications to improve the performance of APR tools.
- The product-based and system-based approaches for repairing variability bugs in the source code of SPL systems. The implementation the proposed approaches can be found at: *https://github.com/ttrangnguyen/SPLRepair*.
- Extensive experimental evaluations showing the performance of the approaches.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background and reviews the related studies. The proposed approach for detecting *false-passing* products is introduced in Chapter 3. The proposed approach for localizing variability faults is described in Chapter 4. Chapter 5 shows the product-based and system-based approaches for repairing variability faults in SPL systems. Finally, Chapter 6 summarizes and concludes this dissertation.

Chapter 2

Background and Literature Review

This chapter introduces background and the concepts which are used in the following sections of the dissertation. First, this chapter introduces the key concepts of the SPL systems, the main testing methodologies, FL and APR techniques. Next, this chapter reviews the related works. Finally, this chapter introduces the popular benchmarks for evaluating testing and debugging approaches of the SPL systems.

2.1 Background

2.1.1 Software Product Line

Traditional single-software engineering targets developing a single product. For each individual software product, developers collect requirements, design, and implement the product. Meanwhile, for SPL engineering, instead of analyzing and implementing a single product each, developers target a *variety of products* that are *similar* but *not identical* [1]. For this purpose, the development process of SPL systems considers two important factors: *variability* and *reuse*. Figure 2.1 illustrates the overview process of developing an SPL system. There are two main processes: Domain engineering and Application engineering.

Domain engineering analyzes the domain of a product line and develops reusable artifacts. This process does not implement any specific product, yet it develops features that can be used in multiple products. Features are the solutions for the requirements and problems of the stakeholders.

Application engineering focuses on developing a specific product tailored to the needs of a particular customer. This process is similar to the development process of traditional single-system, but reuses features from domain engineering. For a customer's requirements, the suitable features of the system are selected and combined to derive a product.

Overall, an SPL is a product family that consists of a set of products sharing a common code base. These products distinguish from the others in terms of their *features* [1].

Definition 2.1 ***(Software Product Line System).*** *A **S**oftware **P**roduct **L**ine System*

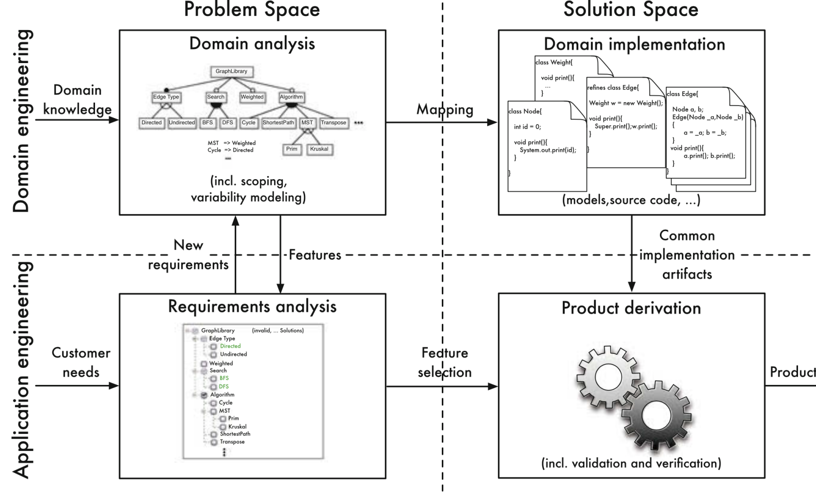


Figure 2.1: Overview of an engineering process for software product lines[1]

(SPL) $\mathfrak{S}$ *is a 3-tuple* $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$*, where:*

- $\mathbb{S}$ *is a set of code statements that are used to implement* $\mathfrak{S}$*,*
- $\mathbb{F}$ *is a set of the features in the system. A feature selection of a feature* $f \in \mathbb{F}$ *is the state of being either enabled (on) or disabled (off)* $(f = T/F$ *for short), and*
- $\varphi : \mathbb{F} \rightarrow 2^{\mathbb{S}}$ *is the feature implementation function. For a feature* $f \in \mathbb{F}$*,* $\varphi(f) \subset \mathbb{S}$ *refers to the implementation of* f *in* $\mathfrak{S}$*, and* $\varphi(f)$ *is included in the products where* f *is on.*

Feature is one of the fundamental interests of SPL engineering. However, the concept of feature is complex and challenging to define precisely. On the one hand, features specify the intentions of the stakeholders of a SPL system. On the other hand, features are used to structure and reuse software artifacts. Thus, there are different variants of feature definition. Following the definition of Apel at al. [1], a feature is a characteristic or end-user-visible behavior of a software system. Features are used in SPL engineering to specify *commonalities* and *differences* of the products of an SPL system.

For an SPL system, the *valid* combination of features are defined by a *feature model*. A feature model of an SPL system has a hierarchical structure which documents all the features of an SPL system and their relationships.

Figure 2.2 shows the feature model of *Elevator* system. This system is implemented by five features, $\mathbb{F} = \{Base, Weight, Empty, TwoThirdsFull, Overloaded\}$. In *Elevator*, *Base*

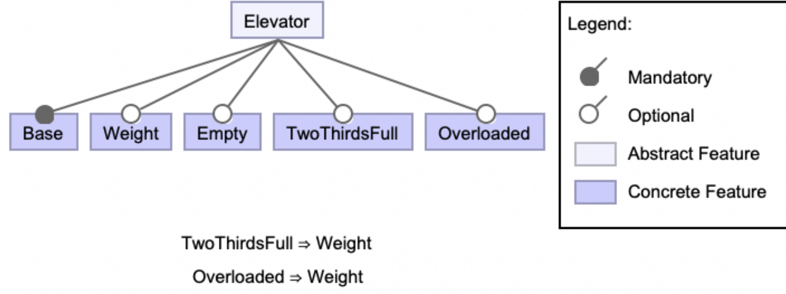


Figure 2.2: An example of feature model of Elevator system

is the mandatory feature implementing the basic functionalities of the system, while the others are optional. In addition, *TwoThirdsFull* is expected to limit the load not to exceed 2/3 of the elevator's capacity, while *Overloaded* ensures the maximum load is the Elevator's capacity. Specifically, *TwoThirdsFull* will block the Elevator when its weight is greater than 2/3 of the allowed capacity. Meanwhile, *Overloaded* will block the Elevator if its weight exceeds the allowed capacity. Both *TwoThirdsFull* and *Overloaded* need information about the total weights of people/things inside the elevator cabin, which is recorded by feature *Weight*. Thus, in an *Elevator* variant where *TwoThirdsFull* and/or *Overloaded* are enabled, *Weight* must also be enabled as specified by the *constraints* in the feature model.

A set of the selections of all the features in $\mathbb{F}$ defines a *configuration*. A configuration which satisfies all the constraints defined by the feature model is a *valid configuration*. Any non-empty subset of a configuration is called a *partial configuration*. A configuration specifies a single *product*. For example, configuration $c_1 = \{Empty = F, Weight = T, TwoThirdsFull = F, Overloaded = F\}$ specifies product p_1. A product is the composition of the implementation of all the enabled features, e.g., p_1 is composed of $\varphi(Base)$ and $\varphi(Weight)$.

The sets of all the possible valid configurations and all the corresponding products of $\mathfrak{S}$ are denoted by $\mathbb{C}$ and $\mathbb{P}$, respectively ($|\mathbb{C}| = |\mathbb{P}|$). In practice, a subset of $\mathbb{C}$, C (the corresponding products $P \subset \mathbb{P}$), is sampled for testing and finding bugs. Unlike non-configurable code, bugs in SPL systems can be *variable* and only cause the failures in certain products.

Definition 2.2 ***(Variability Fault).*** *Given a buggy SPL system $\mathfrak{S}$ and a set of products of the system, P, which is sampled for testing, a variability bug is an incorrect code*

Table 2.1: The sampled products and their overall test results

P	C	*Base*	*Empty*	*Weight*	*TwoThirdsFull*	*Overloaded*
p_1	c_1	T	F	T	F	F
p_2	c_2	T	T	T	F	F
p_3	c_3	T	T	F	F	F
p_4	c_4	T	F	T	T	F
p_5	c_5	T	F	T	T	T
p_6	c_6	T	T	T	F	T
p_7	c_7	T	F	T	F	T

P and C are the sampled sets of products and configurations.
p_6 and p_7 fail at least one test (*failing products*). Other products pass all their tests (*passing products*).

statement of $\mathfrak{S}$ that causes the unexpected behaviors (failures) in a set of products which is a non-empty strict subset of P.

In other words, SPL system $\mathfrak{S}$ contains variability bugs if and only if P is categorized into two separate non-empty sets based on their test results: the *passing products* P_P and the *failing products* P_F corresponding to the *passing configurations* C_P and the *failing configurations* C_F, respectively. Every product in P_P passes all its tests, while each product in P_F fails at least one test. Note that $P_P \cup P_F = P$ and $C_P \cup C_F = C$.

Definition 2.3 *(**Passing product**). Given a product p and its test suite T, p is a passing product if $\forall t \in T$, t is a passed test.*

Definition 2.4 *(**Failing product**). Given a product p and its test suite T, p is a failing product if $\exists t \in T$, t is a failed test.*

Listing 2.1: An example of variability bug in Elevator System

```
1  int maxWeight = 2000, weight = 0;
2
3  //#ifdef Empty
4  void empty(){ persons.clear();}
5  //#endif
6  void enter(Person p){
7      persons.add(p);
8      //#ifdef Weight
9      weight += p.getWeight();
10     //#endif
```

```
11  }
12  void leave(Person p){
13      persons.remove(p);
14      //#ifdef Weight
15      weight -= p.getWeight();
16      //#endif
17  }
18  ElevState stopAtAFloor(int floorID){
19      ElevState state = Elev.openDoors;
20      boolean block = false;
21      for (Person p: new ArrayList<Person>(persons))
22          if (p.getDestination() == floorID)
23              leave(p);
24      for (Person p : waiting) enter(p);
25      //#ifdef TwoThirdsFull
26      if (weight >= maxWeight*2/3)
27          block = true;
28      //#endif
29      //#ifdef Overloaded
30      if(block == false){
31          if (weight == maxWeight )
32              //Patch: weight >= maxWeight
33              block = true;
34      }
35      //#endif
36      if (block == true)
37          return Elev.blockDoors;
38      return Elev.closeDoors;
39  }
```

Listing 2.1 shows a simplified variability bug in *Elevator* system [5]. The overall test results of the sampled products are shown in Table 2.1. In Listing 2.1, the bug (incorrect statement) at line 31 causes the failures in products p_6 and p_7.

In this system, the implementation of *Overloaded* (lines 30–34) does not behave as specified. If the total loaded weight (`weight`) of the elevator is tracked, then instead of blocking the elevator when `weight` exceeds its capacity (`weight >= maxWeight`), its actual implementation blocks the elevator **only** when `weight` is equal to `maxWeight` (line 31). Consequently, if *Weight* and *Overloaded* are on (and *TwoThirdsFull* is off), even the total loaded weight is greater than the elevator's capacity, then (`block==false`) the elevator still dangerously works without blocking the doors (lines 37–39).

This bug (line 31) is *variable* (*variability bug*). It is revealed not in all the sampled products, but only in p_6 and p_7 (Table 2.1) due to the *interaction* among *Weight*, *Overloaded*, and *TwoThirdsFull*. Specially, the behavior of *Overloaded* which sets the value of *block* at line 33 is interfered by *TwoThirdsFull* when both of them are on (lines 27 and 30). Moreover, the incorrect condition at line 31 can be exposed only when *Weight* $= T$, *TwoThirdsFull*$=F$, and *Overloaded* $= T$ in p_6 and p_7. In Table 2.1, $P_P = \{p_1, p_2, p_3, p_4, p_5\}$, and $P_F = \{p_6, p_7\}$.

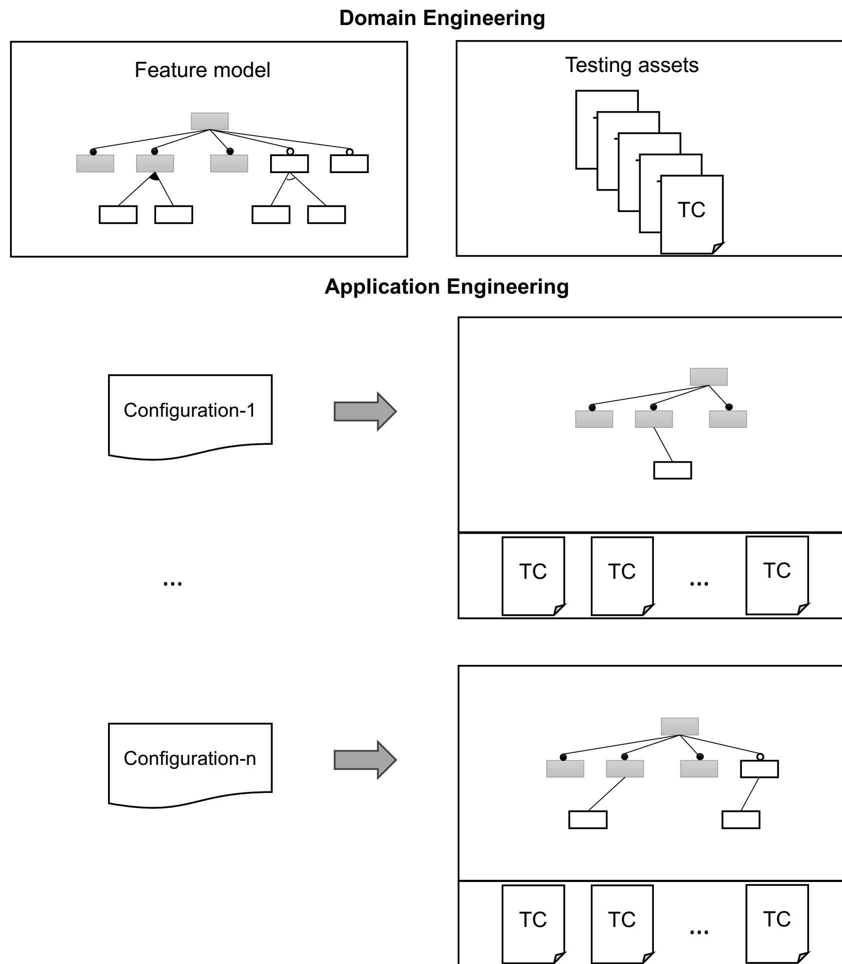


Figure 2.3: SPL testing interest: actual test of products [2]

2.1.2 Testing Software Product Lines

In an SPL system, *features* are fundamental building blocks for specifying products. All possible products of the system are defined by the *feature model*, which represents the dependencies and relationships among features. Guaranteeing the quality of the SPL system means assuring not only every feature of the system works as expected but also that the combinations of the features will work correctly as well [2].

Figure 2.3 shows the testing procedure on end-product functionality. The *domain engineering* defines features, feature model, and testing asserts (e.g., test cases, test scenarios), etc. In the *application engineering*, a concrete product is created by selecting a specific set of features. When a product is instantiated, test cases are selected and concretized according to the product's requirements. After that, each product is validated against its own selected test suite.

```
#ifdef A
  // code 1
#endif

#ifdef B
  // code 2
#else
  // code 3
#endif

#ifdef C
  // code 4
#endif
```

pair-wise			
config-1:	!A	!B	C
config-2:	!A	B	!C
config-3:	A	!B	!C
config-4:	A	B	C

one-enabled			
config-1:	A	!B	!C
config-2:	!A	B	!C
config-3:	!A	!B	C

one-disabled			
config-1:	!A	B	C
config-2:	A	!B	C
config-3:	A	B	!C

most-enabled-disabled			
config-1:	A	B	C
config-2:	!A	!B	!C

statement-coverage			
config-1:	A	B	C
config-2:	A	!B	C

Figure 2.4: Example of sampling algorithms [3]

However, due to the variability inherent to the SPL systems, developers often need to consider a vast number of configurations when they execute tests or perform static analysis [3]. As the configuration space often explodes exponentially with a large number of configuration options, it is infeasible to test and analyze every individual product of a real-world SPL system. For example, with +12K compile-time configuration options, the Linux Kernel can be generated to billions of variants. Thus, testing all the possible variants/products of the Linux Kernel is impossible.

In practice, to systematically perform QA for an SPL system, products are often selected according to several *configuration selection* strategies. The most popular strategies include the sampling algorithms which achieve feature interaction coverage such as *combinatorial interaction testing* [51–53], *one-enabled* [3], *one-disabled* [14], *most-enabled-disabled* [54], or *statement-coverage* [33], etc to reduce the number of configurations. Each sampling algorithm is explained by using the example snippet in Figure 2.4.

The *combinatorial interaction testing* or *t-wise* algorithm [51–53] aims to systematically reduce the number of tested products while maximizing the coverage of possible interactions between system features. The intuition is that various failures of SPL systems are caused by the undesirable interactions among the features. Thus, the testing process should cover as many feature interactions as possible to increase the detected faults.

In particular, *pair-wise* ($t = 2$) checks all pairs of configuration options. For three features A, B, and C in Figure 2.4, there are a total of 12 pairs of configuration options such as (A, B), $(!A, B)$, $(A, !B)$, $(!A, !B)$, etc. To cover all of these pairs of configuration options, this sampling algorithm selects four configurations as shown in Figure 2.4. Considering

options A and B, there is a configuration where both options are disabled (config-1), two alternative configurations where only one of them is enabled (config-2 and config-3), and another configuration where both configuration options are enabled (config-4). The same situation occurs for configuration options A and C, and B and C.

Similarly, for t with the other integer values such as *three-wise* ($t = 3$) selects configurations covering all the possible combinations of any three features and *four-wise* ($t = 4$) selects configurations covering all the possible combinations of any four features of the system. In general, the $t - wise$ algorithm selects a minimal set of configurations that covers all t combinations of features. The larger t, the larger the size of the sample set.

The *statement-coverage* algorithm [33] selects configurations where each optional feature is enabled at least once. In other words, this algorithm aims to select configurations such that each statement (implementing features) of the system is validated at least once in a product. For example, by enabling all configuration options A, B, and C in config-1, code blocks `code 1`, `code 2`, and `code 4` are selected. However, by only this configuration, the code block `code 3` has not been selected. With config-2, A and C are enabled and B is disabled, the code blocks `code 1`, `code 3`, and `code 4` are selected. Thus, to guarantee that each code block is tested at least once, both config-1 and config-2 are selected by the statement-coverage algorithm.

The *most-enabled-disabled* algorithm [54] checks two samples independently. One configuration aims to enable as many options as possible. In contrast, the other aims to disable as many options as possible. For example, if there are no constraints among configuration options, this algorithm selects to test two configurations as shown in Figure 2.4. Config-1 enables all three options, and config-2 disables all of them.

The *one-disabled* algorithm [14] selects samples by disabling one configuration option at a time. Meanwhile, the *one-enabled* algorithm [3] selects samples by enabling one configuration option at a time. As shown in Figure 2.4, the one-disabled algorithm disables A in config-1, B in config-2, and C in config-3. In contrast, the one-enabled algorithm alternatively enables one of these configuration options in each configuration.

Moreover, several approaches about *configuration prioritization* [15, 55, 56] have been proposed to improve the testing productivity. For example, Al-Hajjiaji et al. [55, 56] select the configurations for testing based on the similarity of the configurations with the previously selected ones. Nguyen et al. [15] prioritize configurations based on their number of potential bugs, which are measured by analyzing the feature interactions.

	t_1	t_2	...	t_m
c_1	a_{11}	a_{12}	...	a_{1m}
c_2	a_{21}	a_{22}	...	a_{2m}
...	...	...	...	...
c_n	a_{n1}	a_{n2}	...	a_{nm}
result	r_1	r_2	...	r_m

Figure 2.5: Program spectrum of a program with n elements and m test cases

2.1.3 Fault Localization

Although testing could help discover faults due to the observed erroneous behaviors, finding and fixing them is an entirely different matter. Fault localization, identifying the locations of program faults, is critical in program debugging, yet widely recognized as a tedious, time-consuming, and prohibitively expensive activity [25]. For effective and efficient fault finding, multiple FL approaches for partially or fully automated figuring out the positions of the faults have been proposed. These FL approaches are often categorized into eight groups according to their techniques, including slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based, and miscellaneous techniques.

Amongst these techniques, Spectrum-Based Fault Localization (SBFL) is considered the most prominent due to its lightweight, efficiency, and effectiveness [57]. Specifically, SBFL is a dynamic program analysis technique that leverages the testing information (i.e., test results and code coverage) for measuring the suspiciousness scores of the code components such as statements, basic blocks, methods, etc. The intuition is that, in a program, the more failed tests and the fewer passed tests executed by a code component, the more suspicious the code component is. The component with the higher suspiciousness score is more likely to be buggy.

In particular, an SBFL technique first runs tests on the target program and records the *program spectrum*, which are the run-time profiles about which program components are executed by each test. Then, the suspiciousness scores of program components are assessed based on the recorded program spectrum and the test results (i.e., passing or failing). There are various SBFL formulae have been proposed for calculating suspiciousness scores.

The program spectrum of a program having n components and tested by m test cases are

Table 2.2: Several popular SBFL formulae [4]

SBFL	Formulae
Tarantula [17]	$S(c) = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$
Ochiai [58]	$S(c) = \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$
Op2 [29]	$S(c) = e_f - \frac{e_p}{e_p+n_p+1}$
Barinel [59]	$S(c) = 1 - \frac{e_p}{e_p+e_f}$
Dstar2 [60]	$S(c) = \frac{(e_f)^2}{e_p+n_f}$

shown in Figure 2.5. Particularly, the program spectrum of this program is a matrix A size $n \times m$ where each column specifies the execution profile of a test case, and each row indicates whether a component is executed by tests. For instance, a_{ij} specifies whether component c_i is executed by test t_j. $a_{ij} = 1$ means c_i is executed by t_j, and $a_{ij} = 0$ otherwise. The pass/fail information of tests is stored in a vector r, the result/error vector, where r_j signifies whether test t_j has passed ($r_j = 0$) or failed ($r_j = 1$).

The pair $\langle A, r \rangle$ is the input for SBFL, which measures the statistical *similarity coefficient* between the vector r and the activity profile of each component c_i, i.e., vector $A[i]$. There are various SBFL formulae have been proposed for calculating such similarity coefficients, such as Tarantula [17], Ochiai [58], Op2 [29], Barinel [59], and Dstar2 [60]. Their formulae are listed in Table 2.2, where e_f and e_p are the numbers of failed and passed tests executing the program component c, while n_f and n_p are the numbers of failed and passed tests that do not execute this component.

Figure 2.6 illustrates an example of program spectrum and the FL results of two SBFL metrics, Tarantula and Ochiai. As seen, the target program is `mid`, which finds the middle value among three inputs. Statement s_7 is a buggy statement that incorrectly assigns the value of `y` to `m` instead of assigning the value of `x` to `m`. This function is tested by 6 test cases in which one test failed and the others passed. By both Tarantula and Ochiai, the buggy statement s_7 has the highest suspiciousness score, which should be prioritized to investigate by developers to find and fix the bug.

	Program	Spectra						SBFL Result	
1	**int** mid(int x, int y, int z) {	mid(3, 3, 5)	mid(1, 2, 3)	mid(3, 2, 1)	mid(5, 5, 5)	mid(5, 3, 4)	mid(2, 1, 3)	Tarantula	Ochiai
2	**int** m = z;	1	1	1	1	1	1	0.50	0.41
3	**if** (y < z)	1	1	1	1	1	1	0.50	0.41
4	**if** (x < y)	1	1	0	0	1	1	0.63	0.50
5	m = y;	0	1	0	0	0	0	0.00	0.00
6	**else if** (x < z)	1	0	0	0	1	1	0.71	0.58
7	m = y; //bug: m = x;	1	0	0	0	0	1	0.83	0.71
8	**else**	0	0	1	1	0	0	0.00	0.00
9	**if** (x > y)	0	0	1	1	0	0	0.00	0.00
10	m = y;	0	0	1	0	0	0	0.00	0.00
11	**else if** (x > z)	0	0	0	1	0	0	0.00	0.00
12	m = x;	0	0	0	0	0	0	0.00	0.00
13	**return** m;	1	1	1	1	1	1	0.50	0.48
14	}								
		Passed	Passed	Passed	Passed	Passed	Failed		

Figure 2.6: Example of program spectrum and FL results by Tarantula and Ochiai

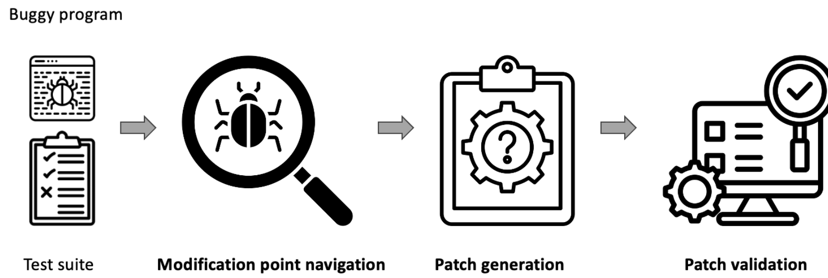


Figure 2.7: Standard steps in the pipeline of the test-suite-based program repair

2.1.4 Automated Program Repair

To reduce the cost of software maintenance, multiple APR techniques have been proposed in the past. The most popular APR approach is *test-suite-based program repair* [40, 61, 62], such as GenProg [18], Nopol [37], and Cardumen [63], which use test suites as the specification of the program's expected behaviors. For repairing a program failed by at least one test, these APR approaches attempt to generate candidate patches. Then, the available test cases are used to check whether the generated patches can fix the program.

In practice, the test-suite-based program repair tools are commonly implemented in three steps, as shown in Figure 2.7. First, code elements of the program under repair are selected as the positions for attempting to fix by the **modification point navigation** step. In this step, to narrow down the search space, an FL technique can be applied to detect and rank suspicious code elements according to their suspiciousness. Then, the probability of being selected of the code elements is often decided based on their suspiciousness scores. Next, the **patch generation** step generates candidate patches for the selected code positions. A patch can be generated by multiple different techniques.

For example, GenProg [18] generates patches by using existing code from the program under repair, or Nopol [37] collects running time information to build repair constraints and then uses a constraint solver to synthesize patches. Finally, a patch is validated by the test suites of the program to check whether the patched program meets the expected behaviors (**patch validation**).

The concepts of APR including Modification point (Definition 2.5), Modification operator (Definition 2.6), Modification operation (Definition 2.7), and Candidate patch (Definition 2.8) used in this dissertation are formally defined as follows:

Definition 2.5 *(**Modification point**). A modification point $mp = (pos, c_o)$ is a code element that can be modified to repair the buggy program, in which pos is the position of the code element in the program under repair and c_o is its associated (original) code.*

Listing 2.2: An example of buggy code snippet

```
1 public int getGrade(int matrNr) throws ExamDataBaseException{
2   int i = getIndex(matrNr);
3   if(students[++i] != null && !students[i].backedOut){
4     //Patch:  if(students[i] != null && !students[i].backedOut)
5     return pointsToGrade(students[i].points, 0);
6   }
7   throw new ExamDataBaseException("Matriculation number not found");
8 }
```

For example, in GenProg [18, 36], which repairs the program at the statement level, a modification point is a suspicious statement in the program. For instance, with the buggy code in Listing 2.2, a modification point of GenProg could be any suspicious statement in this code, such as $mp = (s_3,$ `if(students[++i] != null && !students[i].backedOut)`$)$. Instead, in Cardumen [63], which fixes the program at the expression level, a modification point is an expression in a suspicious statement. For instance, in the suspicious statement s_3 in Listing 2.2, each of its expressions could be a modification point in Cardumen, such as $mp = (s_3,$ `students[++i] != null`$)$.

Definition 2.6 *(**Modification operator**). A* modification operator *op is the action of transforming a code element into another. In this dissertation, the considered operators are $op \in \{rem, rep, ins_bef, ins_aft\}$, where rem, rep, ins_bef, and ins_aft are remove, replace, insert before, and insert after operators, respectively.*

For a modification point mp, a modification operator can be applied to transform the source code at this point. Namely, the operator *rem* removes code at mp, the operator *rep* replaces the code at mp with a new code, the operator *ins_bef* inserts a new code before mp, while *ins_aft* inserts a new code after mp. To generate the new code for applying insert/replace operators, several approaches [18, 38, 50, 63] leverage the *ingredients* from the program under repair or from the other projects. Instead, other approaches synthesize new code without using ingredients, such as jMutRepair [36] or Nopol [37].

Definition 2.7 *(**Modification operation**). Given a modification point* $mp = (pos, c_o)$*, a* modification operation $d = op(mp, c_n)$ *is the transformation from the original code* c_o *to a new code by applying the repair operator op with the code* c_n *at the position pos. In particular, the transformation of each modification operator is defined as follows:*

- $rem(mp, c_n) = (pos, ``")$,
- $rep(mp, c_n) = (pos, c_n)$,
- $ins_bef(mp, c_n) = (pos, c_n + c_o)$, *and*
- $ins_aft(mp, c_n) = (pos, c_o + c_n)$.

Definition 2.8 *(**Candidate patch**). A* candidate patch *(or* patch *for short) is the transformation result of a list of one or more modification operations.*

In general, a patch could consist of one or more modification operations since a buggy program could be fixed by modifying one or several code statements. A *valid patch* is a candidate patch which passes all the available test cases of the program. Originally, the number of valid patches was a common metric to measure the performance of APR tools [18, 64]. However, a test suite is often weak and inadequate [65–68], and it cannot cover all the behaviors of the program. Therefore, despite passing all the available test cases, a patch could still break other behaviors or introduce new faults, which are not covered by the given test suite [67]. Such a valid patch is then referred to as a *plausible patch* or *test-adequate patch*, which needs to be further manually investigated by developers to ensure its correctness.

2.2 Literature Review

This section comprehensively reviews studies investigating the inherent characteristics and critical problems of SPL systems, i.e., feature interaction and variability bugs. This dissertation also critically examines the studies about QA for SPL systems with an emphasis on methodologies and best practices to ensure the reliability and performance of this kind of system. Moreover, testing is a common and effective method for assuring the quality of the systems and the test results could provide valuable guidance for the debugging process. Thus, test suite quality is an important problem. This section examines the studies that measure the effectiveness of the test suites and improve the test suite reliability by addressing the coincidental correctness problem. Finally, this section analyzes the existing approaches for localizing and repairing bugs in both non-configurable and configurable systems.

Feature interaction in SPL system. There are a large number of approaches to investigate feature interactions as the influences of the features on the others' behaviors in an unexpected way [12, 69]. Depending on the level of granularity and purposes, various approaches were proposed to detect feature interactions. In black-box fashion, Siegmund et al. [70] detects interactions to predict system performance by analyzing the influences of the selected features on the others' non-functional properties. Based on the specifications, model checking is also used to check whether the combined features hold the specified properties [71, 72]. In several studies [15, 73, 74], interactions are detected by analyzing code. Nguyen et al. [15] detect interactions based on their shared program entities. In other studies [73, 74], they leverage control and data flow analyses to identify the interactions among features. iGen [75] employs an iterative method that runs the system, captures coverage data, processes data to infer interactions, and then creates new products to further refine interactions in the next iteration.

Variability faults. Variability faults are complex and difficult to be detected by both humans and tools because they involve multiple system features and only be revealed in certain products [12, 14, 76, 77]. Abal et al. [14, 76] analyzed the real variability bugs in several large highly-configurable systems such as Linux kernel and Busybox to understand the complexity and the nature of this kind of bugs. In order to make a white-box understanding of interaction faults, Garvin et al. [12] presented the criteria for interaction faults to be present in systems. Their criteria are about the statements whose (non-) execution is necessary for the failure to be exposed/masked.

Quality assurance for SPL system. Configurable systems create a mechanism to flexibly tailor products to customers' needs. Unfortunately, the large number of features, as well as their mutual interactions make their quality become notoriously difficult to assure. There are various studies about variability-aware analysis [78, 79] for the purpose of type checking [80, 81], testing [82, 83], control/data flow analysis[81, 84], and performance prediction [85, 86].

In addition, testing an SPL system [87, 88] is a complex and costly task since the variety of the interactions of system features and a large number of derived products. A large number of studies have been conducted, and various testing strategies have been proposed. To efficiently assure software quality, various *sampling algorithms* have been introduced [19, 20, 89–93]. In addition, to improve the efficiency of the testing process, several approaches about *configuration selection* [2, 94] and *configuration prioritization* [15, 55, 95] have also been proposed. Several other approaches are aimed for testing for configurable systems [94, 96].

Test suite effectiveness measurement. Various metrics have been proposed to measure the quality of the test suites. Specially, code coverage is one of the most popular metrics, which measures the percentage of code elements (e.g., statements, branches, decisions, etc.) covered by test suites [97]. Also, Perez et al. [98] proposed Density-Diversity-Uniqueness (DDU) metric which aims at measuring the effectiveness of the suites in term of applying SBFL to detect faults in the corresponding source code. Gonzalez-Sanchez et al. [99] employed information gain algorithm to predict the efficiency of the test suites based on the system's size, the coverage density, and the uniform of coverage distribution. Baudry et al. [100] proposed a test criterion based on the Dynamic Basic Block, the test data (traces), and the software control structure to evaluate the fault localizing capacity of the test cases. Besides, mutation testing techniques [101, 102] are also proposed to evaluate the effectiveness of the test suites in detecting mutated faults.

Coincidental correctness detection and impact mitigation. Coincidental correctness has been proven as a prevalent problem in software testing [21]. Also, practical experiments have been conducted to demonstrate that this problem adversely affects FL performance [21, 103, 104]. Many techniques have been proposed to detect coincidentally passed tests [21, 105–107], which execute the faults, yet produce correct outputs. After that, they cleansed the test suites from these detected unreliable tests to enhance FL performance. Bandyopadhyay et al. [108] proposed an approach to predict and weight

coincidentally passed tests to calculate the suspiciousness scores. In general, these approaches investigate each individual passed test case to detect coincidental correctness and boost FL performance in non-configurable code.

Fault localization. There are various approaches proposed to identify the locations of faults in programs [25, 109–113]. Program slicing [26, 27] is used in many studies to reduce the search space while localizing bugs by deleting irrelevant parts in code. Both static slicing [109, 110] and dynamic slicing [111, 114] are used to aid programmers in finding defects. In addition, by SBFL, a program spectrum, which records the execution information of a program, can be used to localize bugs. This idea was suggested by Collofello and Cousins [115]. To calculate suspiciousness, in early studies [116, 117], only information of failed tests was used. In later studies [31, 118, 119], which are better, both of the passed and failed tests are utilized. Moreover, several studies [120, 121] have shown that the FL performance is improved by combining SBFL technique with slicing methods.

Automated program repair. In recent years, APR has attracted a lot of attention from both industry and academia. Various APR techniques have been proposed. Goues et al. [122] divided the APR techniques into three main groups: heuristic-based [18, 50, 123–126], constraint-based [37, 127], and learning-based repair [38, 39, 128–131].

In *heuristic-based repair* direction, a search strategy such as random search [123], genetic programming [18], or multi-objective genetic programming [124] is leveraged to guide the search of valid patches. For example, GenProg [18] is one of the most well-known program repair tools which uses a genetic programming technique to guide the patch generation process. This tool has been demonstrated to be able to fix real bugs in non-trivial programs. However, since the genetic algorithm needs to measure fitness values and distinguish better and worse patches, one general problem of APR tools using this technique is that they are computationally expensive. Besides, Qi et al. [123] introduce RSRepair, in which a patch is generated by a random-search technique which is far less complicated compared to a genetic algorithm. Their experiments have shown that RSRepair is more efficient and effective than GenProg. Recently, various advanced APR tools in this search-based direction have been introduced, and they have shown their high performance, such as SimFix [125], SSFix [126], Arja [124], and TransplantFix [50].

In *constraint-based repair* direction, repair constraints that the patched code should satisfy, are constructed. The patches are generated by solving such constraints. For example, SemFix [127] is a representative constraint-based APR. This tool generates repair

constraints based on the expression in the buggy location using (controlled) symbolic execution program synthesis. In another research, Xuan et al. [37] introduced Nopol, which collects runtime information, including variable and actual values, to construct constraints and then uses a Satisfiability Modulo Theories (SMT) solver to synthesize patches for fixing buggy conditional statements.

In *learning-based repair* direction, the APR tools leverage machine learning/deep learning algorithms to learn from a large corpus of existing (correct) patches and generate candidate patches for a newly encountered program. For instance, DeepRepair [128] uses code similarities, which are reasoned by a deep learning model, to select and transform ingredients. In addition, DEAR [38] adopts tree-based Long Short-Term Memory (LSTM) models and uses a divide-and-conquer strategy to learn proper code transformations. It can generate fixes by modifying one or several statements at the same time.

For *repairing buggy* SPL systems, Arcaini et al. [42] have proposed a solution to repair *variability models* to prevent incorrect configurations from being generated in the solution space. Besides, the wrong configurations, which violate the model constraints, are solved and fixed by Xiong et al. [44]. In that work, they use a constraint solver to automatically generate range fixes and ensure the desired properties of the generated fixes. Furthermore, Echeverría et al. [35] realize the importance and complexity of bug-fixing in the SPL system. They conducted an empirical study to analyze the patches of industrial SPL systems, which were manually fixed by engineers. This study also confirms that fixing variability bugs is challenging, especially propagating the fix when the source of the bug is the interaction between features.

2.3 Benchmarks for Software Product Lines

For evaluating QA approaches of the SPL systems, Abal et al. [76] and Mordahl et al. [132] constructed and published the datasets of real-world variability bugs. Their datasets contain 98 variability bugs collected from bug-fixing commits in the Linux, Apache, BusyBox, and Marlin repositories. Importantly, to support researchers to understand the bugs and efficiently use the bugs for developing and evaluating their QA tools, Abal et al. [76] manually analyze the bugs to create self-contained simplified versions and simplified patches. However, these bugs are not provided along with corresponding test suites, and most of these bugs are compile-time bugs.

In addition, Arrieta et al. [6] also constructed a set of artificial bugs to evaluate their approach to localizing bugs in SPL systems. They selected nine SPL systems of different sizes; seven are taken from SPLOT Repository [133] and two real-world systems, i.e., the Drupal framework and Unmanned Aerial Vehicle. However, due to the unavailability of the feature models, source code, and test cases, as well as the time-consuming test execution process, they resorted to a fault simulator in eight of the systems (where no code nor test cases were available) except for the Unmanned Aerial Vehicle.

For the simulation of faults, they developed a fault generator that receives a feature model as an input and returns a random list of faulty feature sets as an output. They aim to simulate different numbers and types of faults in the SPL systems under test. In addition, the test results are also simulated with the assumption that if a product contains any of the features labeled as faulty, the execution of the product is classified as failed; otherwise, it is classified as successful. However, their dataset has not been published.

Furthermore, Ngo et al. [5] propose a large dataset of variability faults found by testing. In this dataset, to generate a large number of variability bugs, the bug generation process includes three main steps: Product Sampling and Test Generating, Bug Seeding, and Variability Bug Verifying. First, for an SPL system, a set of products is systematically sampled by the existing techniques [3]. To inject a fault into the system, a random modification is applied to the system's original source code by using a mutation operator. Finally, each generated bug is verified against the conditions of the variability bug to ensure that the fault is a variability bug and is caught by the tests. The detailed design decisions can be found in [5].

The dataset overview is shown in Table 2.3. In total, there are 1,570 buggy versions of the six subject SPL systems. These systems are selected of different sizes of features and code statements. Among them, 338 versions contain a single bug each, while 1,232 versions have two or more bugs. There are about 70% of the bugs contained in the Assignment or Conditional statements.

For testing, each buggy SPL system is sampled with 4-wise coverage. In general, for an SPL system, the number of sampled products depends not only on the number of features but also on the feature model of the system. For instance, although ExamDB and BankAccountTP have the same number of features, to achieve 4-wise coverage by sampling technique, BankAccountTP needs to generate 34 products. In comparison, this figure for ExamDB is only eight products. Moreover, there are 5/6 systems whose

Table 2.3: Dataset Statistics [5]

System	**Details**		**Test info**			**Bug info**		
	#LOC	#F	#SP	#Tests	Cov	Single-Bug	2-Bug	3-Bug
ZipMe	3460	13	25	255.0	42.9	55	120	129
GPL	1944	27	99	86.9	99.4	105	190	77
Elevator-FH-JML	854	6	18	166.0	92.9	20	41	61
ExamDB	513	8	8	133.3	99.5	49	126	88
Email-FH-JML	439	9	27	86.0	97.7	36	34	56
BankAccountTP	143	8	34	19.8	99.9	73	238	72

#F and **#SP** stand for the number of features and the average sample size.

Cov stands for the statement coverage (%).

#IF stands for the average number of the involving features.

generated test suite reaches +90% statement coverage, and three of them almost reached 100% statement coverage. Especially due to a large code base, ZipMe has 255 tests per product, but its statement coverage only stays at 42.9%.

To the best of our knowledge, this is the only public dataset containing the versions of the SPL systems that failed by variability bugs found through testing. Thus, the dissertation employs this dataset [5] for evaluating the approaches of detecting *false-passing* products, localizing, and fixing variability faults.

Chapter 3

False-passing Product Detection

This chapter introduces CLAP, a novel approach to detect *false-passing* products in buggy SPL systems. First, this chapter shows the motivation and formulate the problem of detecting *false-passing* products in the SPL system. Then, this chapter introduces a *false-passing* products detection approach and discusses the strategies to mitigate the impact of these products on variability fault localization.

3.1 Introduction

Thorough testing is generally required to guarantee the quality of programs. However, it is often hard, tedious, and time-consuming to conduct thorough testing in practice. Various bugs could be neglected by the test suites since it is extremely difficult to cover all the programs' behaviors. Moreover, there are kinds of bugs which are challenging to be detected due to their difficulties in infecting the program states and propagating their incorrectness to the outputs [22]. Consequently, even when the defects is reached, there are test cases that still obtain correct outputs, i.e., *coincidentally correct/passed* tests. Indeed, coincidental correctness is a prevalent problem in software testing [21], and this phenomenon causes a severely negative impact on FL performance [21, 23, 24].

Similar to testing in non-configurable code, the coincidental correctness phenomenon also happens in SPL systems and causes difficulties in finding faults in these systems. Specifically, for an SPL system, a set of products is often sampled for testing. Each sampled product is composed of a set of features of the system and tested individually by its test suite as a singleton program. For a buggy SPL system, the bugs could be in one or more products. Ideally, if a product contains bugs (*buggy products*), the bugs should be revealed by its test suite. In other words, there should be at least a failed test after testing. However, if the test suite of a buggy product is ineffective in detecting the bugs, the product's overall test result will be passing. For instance, the suite does not cover the product's buggy statements or those test cases could reach the buggy statements but could not propagate the incorrectness to the outputs, the product still passes all the tests. Such a passing product is indeed a buggy product, yet incorrectly considered as passing.

That passing product is namely a *false-passing* product. Due to the unreliability of test results, these *false-passing* products might negatively impact the FL performance. In particular, the performance of two main SBFL strategies in SPL systems, *product-based* and *test case-based*, is affected.

First, the *product-based* FL techniques [6] evaluate the suspiciousness of a statement in a buggy SPL system based on the appearance of the statement in failing and/or passing products. Specially, the key idea to find bugs in an SPL system is that a statement which is included in more failing products and fewer passing products is more likely to be buggy than the other statements of the system. Misleadingly counting a buggy product as a passing product incorrectly decreases the number of failing products and increases the number of passing products containing the buggy statement. Consequently, the buggy statement is considered less suspicious than it should be.

Second, the *test case-based* FL techniques [25] measure the suspicious scores of the statements based on the numbers of failed and passed tests executed by them. Indeed, *false-passing* products could lead to under-counting the number of failed tests and over-counting the number of passed tests executed by the buggy statements. The reason is that *false-passing* products contain bugs, but there is no failed test. In these *false-passing* products, the buggy statements are not executed by any test, or they are reached by several tests, yet those tests *coincidentally* passed. Both low coverage test suite and coincidentally passed tests can cause inaccurate evaluation for the buggy statements.

This chapter introduces CLAP, a novel *false-passing* product detection approach for SPL systems that failed by variability bugs. The intuition of the proposed approach is that for a buggy SPL system, the sampled products can share some common functionalities. If the unexpected behaviors of the functionalities are revealed by the tests in some (failing) products, the other products having similar functionalities are likely to be caused failures by those unexpected behaviors. In CLAP, *false-passing* products can be detected based on the *failure indications* which are collected by reviewing the *implementation* and *test quality* of the failing products. To evaluate the possibility that a passing product is a *false-passing* one, CLAP proposes several *measurable attributes* to assess the strength of these failure indications in the product. The stronger indications, the more likely the product is *false-passing*.

The proposed attributes are belonged to two aspects: *product implementation* (products' source code) and *test quality* (the adequacy and the effectiveness of test suites). The at-

tributes regarding *product implementation* reflect the possibility that the passing product contains bugs. Intuitively, if the product has more (suspicious) statements executing the tests failed in the failing products of the system, the product is more likely to contain bugs. For the *test quality* of the product, the *test adequacy* reflects how its suite covers the product's code elements such as statements, branches, or paths [45]. A low-coverage test suite could be unable to cover the incorrect elements in the buggy product. Hence, the product with a lower-coverage test suite is more likely to be *false-passing*. Meanwhile, the *test effectiveness* reflects how intensively the test suite verifies the product's behaviors and its ability to explore the product's (in)correctness [46, 47]. The intuition is that if the product is checked by a test suite which is less effective, its overall test result is less reliable. Then, the product is more likely to be a *false-passing* one.

Furthermore, this chapter discusses several strategies to mitigate the negative impact of *false-passing* products on the performance of the FL approaches. Since the negative impact is mainly caused by the unreliability of the test results, the goal of CLAP is to improve the *reliability* of the test results by enhancing the test quality based on the failure indications. Moreover, the reliability of test results could also be improved by disregarding the unreliable test results at either product-level or test case-level.

This dissertation conducted several experiments on a large dataset of variability bugs which contains 823 buggy versions of six widely-used SPL systems [5]. Totally, there are 14,191 *false-passing* products and 22,555 *true-passing* products. The experimental results show that CLAP achieves more than 90% *Accuracy* in detecting *false-passing* and *true-passing* products. The capability of CLAP in mitigating the negative impact of *false-passing* products on the FL performance is also evaluated. The experimental result shows that CLAP can greatly mitigate the negative impact of *false-passing* products on localizing variability bugs and help developers find bugs much faster.

3.2 Motivation and Problem Formulation

3.2.1 Motivation

To empirically investigate the impact of *false-passing* products on FL, a preliminary study was conducted on 600 buggy versions of six SPL systems in a dataset of variability bugs [5]. For each buggy version, the existence of *false-passing* products is simulated by modifying the test suites of a random number of failing products. Specially, all the failed tests in

the test suite T of each selected product p are removed. Once all the failed tests in T are removed to create test suite T', the bugs in p revealed by T would not be revealed by T'. As a result, p becomes a *false-passing* product with test suite T'. After the simulation, each buggy version contains three groups of products: (1) failing products which contain both failed and passed tests; (2) *false-passing* products which were originally failing products, yet their failed tests were removed; and (3) passing products which originally passed all the tests.

This chapter applies two state-of-the-art FL approaches, SBFL [31, 115–119] and VarCop [1] to localize the variability bugs in each system with and without the existence of the (simulated) *false-passing* products. With the existence of *false-passing* products (*With FPs*), testing information of all the three groups of products, i.e., failing products, *false-passing* products, and passing products, are used to measure the statements' suspiciousness. Without the existence of *false-passing* products (*Without FPs*), testing information of only failing products and passing products are used for localizing faults.

Table 3.1 shows the average *Rank* of the buggy statements which are localized by VarCop and SBFL using the five most popular ranking metrics. As seen, *the presence of false-passing products after testing could significantly reduce the performance of the FL techniques.* On average, with the presence of *false-passing* products, the results of both VarCop and SBFL are significantly decreased by 60% and 20%, respectively. For instance, without the existence of the *false-passing* products, by VarCop with the metric Barinel, the bugs can be found after investigating 5 statements. However, due to the presence of the *false-passing* products, this performance is decreased by 60%, i.e., 12 statements need to be investigated to find the bugs. Similar to SBFL with the metric Barinel, the FL performance is decreased by 20%, i.e., the number of statements that need to be examined increased from 8 to 10 statements.

3.2.2 Problem Formulation

After testing, for an SPL system $\mathfrak{S}$, let $P = \{p_1, ..., p_n\}$ be the set of the sampled products. Each product $p_i \in P$ is tested by a corresponding test suite T_i. In general, the system $\mathfrak{S}$ contains variability bugs if and only if P is categorized into two separate non-empty sets based on their overall test results: the *passing products* P_P and the *failing products* P_F,

[1] This is an approach proposed by this dissertation for variability FL. This approach is introduced in Chapter 4.

Table 3.1: Empirical study about the impact of *false-passing* products on variability fault localization performance (in *Rank*)

Ranking Metrics	VarCop		SBFL	
	With *FPs*	Without *FPs*	With *FPs*	Without *FPs*
Tarantula	11.47	6.25	9.92	8.22
Ochiai	7.93	5.05	7.38	5.91
Op2	6.40	5.71	7.31	7.15
Barinel	11.89	5.36	9.91	8.22
Dstar	7.13	4.87	7.36	5.91

$P_P \cup P_F = P$ [12, 91]. Each product in P_F fails at least one test, while every product in P_P passes all the test cases in its test suite. Among the passing products, a *false-passing* product contains bugs and should be a failing product yet has passed all the tests because of its ineffective test suite.

Definition 3.1 *(**False-passing product**). Given a tested product $p \in P$ and its test suite T, product p is a false-passing product if the following conditions are satisfied:*

(i) There exists a statement s in p, such that s can cause failures for p, and

(ii) Product p passed all the test cases in T.

In other words, for *false-passing* product p, the current test suite T of p is ineffective in detecting bugs in p, thus p has not failed any test in T. Additionally, there exists a test suite $T' \neq T$ such that p could fail at least a test in T'. In this case, T' is more *bug-detecting effective* than T, and the test results of T' is more *reliable* than that of T. On the opposite side, *true-passing* products in a tested buggy SPL system can be formally defined as Definition 3.2.

Definition 3.2 *(**True-passing product**). Given a tested product p whose test suite is T, p is a true-passing product if:*

*(i) There does **not** exist a statement s in p, such that s can cause the failures for p, and*

(ii) As a result, product p passed all the test cases in T.

Definition 3.3 *(**False-passing product detection**). Given 4-tuple $\langle \mathfrak{S}, P, \mathcal{T}, \mathcal{V} \rangle$, where:*

- *$\mathfrak{S}$ is a tested SPL system containing variability bugs,*
- *$P = \{p_1, ..., p_n\}$ is the set of n sampled products, $P = P_P \cup P_F$, where P_P and P_F are the sets of passing and failing products of $\mathfrak{S}$,*
- *$\mathcal{T} = \{T_1, ..., T_n\}$ is a set of test suites, where $\forall i \in [1, n]$, $T_i \in \mathcal{T}$ is the test suite of $p_i \in P$, and*
- *$\mathcal{V} = \{V_1, ..., V_n\}$ is the set of the program spectra, where $\forall i \in [1, n]$, $V_i \in \mathcal{V}$ is the program spectrum of p_i with the test suite $T_i \in \mathcal{T}$.*

False-passing product detection is to output the set of the false-passing products in P_P.

As shown in Section 3.2.1, detecting *false-passing* products could help significantly improve the performance of FL techniques. However, *false-passing* products could be very challenging to be detected. Indeed, *false-passing* phenomenon is caused by the ineffectiveness of testing, thus the bugs in the (*false-passing*) products are not revealed. To identify whether or not a passing product is *false-passing*, new test cases can be generated to further test the product. In practice, it could be prohibitively expensive to test all the product's behaviors. Furthermore, although a large number of tests are added, if the product still passes all the newly generated test cases, it still is not able to confirm the product is *false-passing* or *true-passing*. Moreover, another approach in detecting *false-passing* products is to figure out whether the passing products contain the bugs. However, the bugs, which cause failures in the system, have not been identified yet at that point. This means identifying the presence of bugs in the passing products is also problematic. Hence, *false-passing product detection is necessary but challenging.*

3.3 *False-passing* Product Detection

As a *false-passing* product is a product containing bugs but still passed all its test cases, its overall test result, i.e., being a passing product, is unreliable. Hence, determining if a passing product is *false-passing* can be done by examining: (i) *whether or not the product contains any bug* and (ii) *the reliability of the state of passing all its test cases.*

In general, a product is considered *containing a bug* if it contains a buggy statement and the statement's bugginess can be propagated to product's incorrect output(s). To expose the incorrectness, the buggy statement rarely executes solo, it often executes together with the other statements. Such statements are called *bug-involving statements*. Thus, to determine whether the bug is contained in the product, CLAP *examines whether the product contains both the buggy statement and the corresponding set of bug-involving statements.*

If a product contains bugs, yet *passes all its tests*, its test suite could be inadequate [45] and ineffective [46, 47] in exploring the bug(s) in the product. Its test suite is in low coverage of the product's behaviors and could not cover the unexpected ones. Consequently, the product still passed all its test cases and is misleadingly considered as a passing product. Intuitively, the more inadequate and ineffective the test suite, the less reliable the product's overall test result. Hence, to verify the reliability of the product's overall test result, *it is essential to examine the adequacy and effectiveness of the product's test suite.*

In practice, a buggy system could contain multiple bugs, and not all of the bugs are revealed after testing the sampled products. This work focuses on detecting *false-passing* products regarding the bugs which have been revealed by the failed tests in the failing products of the system. The other passing products, which can not be failed by any of these revealed bugs, are considered as *true-passing* products.

Intuitively, for a set of sampled products of the system, the program and the tests of the failing products can provide the indications to examine the passing ones.

The *failure indications* in the failing products are investigated in terms of product implementation (i.e., the existence of *buggy statements* and *bug-involving statements*) and test quality (i.e., *test adequacy* and *test effectiveness*). Next, determining if a passing product is *false-passing* can be done by measuring the strength of these indications in the product.

To evaluate the strength of the failure indications in a passing product, CLAP proposes a set of measurable attributes. The *product implementation* attributes measure the possibility that the product contains buggy statements and corresponding sets of bug-involving statements (Section 3.3.1). The test quality is examined in terms of test adequacy and test effectiveness. The *test adequacy* attributes measure to what extent the suite covers product's elements (Section 3.3.2). The *test effectiveness* attributes examine how each test case verifies the product's behaviors (Section 3.3.3). Overall, if a passing product has a high possibility of containing bug(s) and has a low-quality test suite, the product

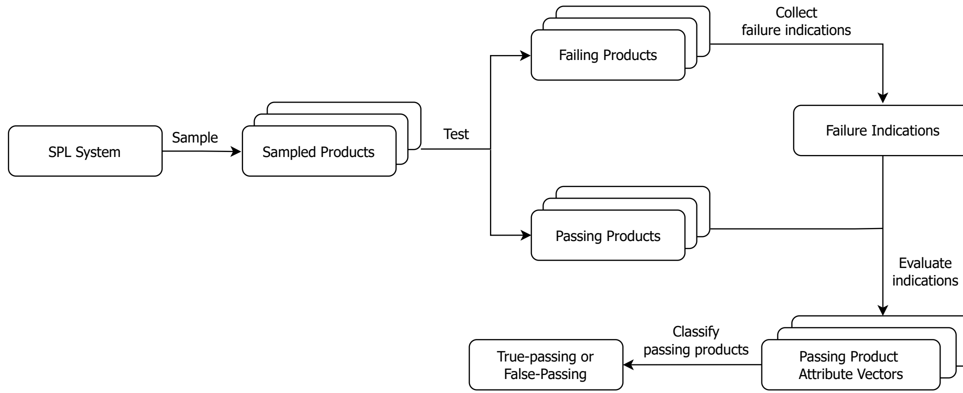


Figure 3.1: CLAP's overview

is more likely to be *false-passing*. The approach overview is shown in Figure 3.1.

To verify the selection of the proposed attributes in detecting *false-passing* products, this chapter conducts experiments on 159 buggy versions of BankAccountTP (so-called *verification dataset*). Overall, there are 1,626 failing products, 1,763 *true-passing* products, and 2,017 *false-passing* products. The construction of this dataset is described in detail in Section 3.5.2. Each attribute's value is measured in each product to confirm that the attribute can distinguish the *true-passing* and *false-passing* products.

3.3.1 Suspiciousness of Product Implementation

To evaluate the failure indications in a passing product p of the buggy SPL system $\mathfrak{S}$ regarding the product's implementation, CLAP investigates the possibility that p contains the *buggy statements* which caused the failures in the failing products of $\mathfrak{S}$. Additionally, the likelihood that p has the statements which involve contributing and propagating the incorrectness of the buggy statements (*bug-involving statements*) is also evaluated while examining the implementation of p.

How possibly does a product contain buggy statements?

For a passing product p, CLAP estimates the possibility that p contains buggy statements by preliminarily measuring the suspiciousness of the statements in p. Intuitively, *if statements in p are highly suspicious, p will be more likely to contain the buggy statements.* The possibility that p contains buggy statements, $bscp(p)$, can be estimated by the total suspicious scores of the statements in that product as shown in Equation 3.1. In this

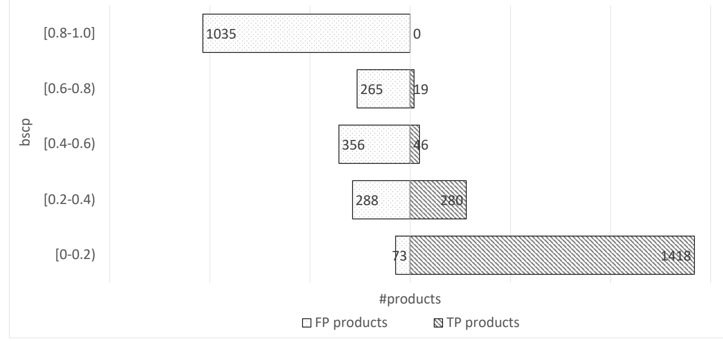


Figure 3.2: The presence of the suspicious statements in the passing products

Equation, $S_p = \{s_1, ..., s_n\}$ is the set of statements of product p, and $\phi(s, P_F, M)$ is the function measuring the suspiciousness score of statement s by using the testing information of the failing products in P_F and the FL technique M. Also, *bscp*(p) is normalized into the range of $[0, 1]$.

$$bscp(p) = normalize\left(\sum_{s \in S_p} \phi(s, P_F, M)\right) \tag{3.1}$$

In *bscp*(p), any FL technique, which can calculate the suspicious score of a statement in an SPL system, can be applied as M in function ϕ. This work uses the testing information of only failing products P_F, whose overall test results are reliable at this point, to measure the suspiciousness of the statements. However, to avoid missing the useful information provided by test results of the passing products, one can use all the sampled products P in the statement suspiciousness evaluation function, ϕ.

Figure 3.2 shows the possibility that passing products contain buggy statements in the *verification dataset*. For each buggy version, CLAP preliminarily measures the suspiciousness of the statements by using SBFL with Op2 on the program spectra of the failing products (function ϕ). As seen, 1418 *true-passing* products (out of 1763 *true-passing* products, about 80%) have the bug-containing possibility less than 0.2. Meanwhile, 1035 *false-passing* products (out of 2017 *false-passing* products, about 50%) have this possibility greater than 0.8. This illustrates that the *false-passing* products often contain a large number of highly suspicious statements. Thus, *the higher bug-containing possibility of a passing product is, the more likely it is a false-passing product.*

How possibly does a product have bug-involving statements?

Bug-involving statements of a buggy statement s are the statements, which impact/be impacted by s. These statements must be executed together with s to expose the incorrectness of s to the outputs. To compute the possibility that a passing product p contains bug-involving statements of s, CLAP measures how similarly s impacting/being impacted in p and a failing product p' containing s. Intuitively, the more similarly s impacting/being impacted in p and p', the more similarly s behaves in these two products. As a result, the higher possibility that s can cause failures in p in a similar way s has caused the failures in p'. This means that p is more likely to be a *false-passing* product.

Meanwhile, the buggy statements have not been found yet at this point. To estimate the possibility that a passing product p contains bug-involving statements, CLAP uses suspicious statements and their impacting/being impacted statements. The suspicious statements are the statements executed by at least a failed test in a failing product. The statements impacting/being impacted by a suspicious statement are suspiciously to be bug-involving statements, so called suspiciously-involving statements.

Let $\mathcal{S} = \{s_1, ..., s_k\}$ be the set of suspicious statements of the given SPL system. For a statement $s \in \mathcal{S}$, let B and B' be the sets of impacting statements which impact s in p and in a failing product p', respectively. Also, F and F' are the sets of being-impacted statements which are impacted by s in p and p'. The similarity of the sets of suspiciously-involving statements of s in p and p' is measured as the similarities of its impacting statements(B and B') and being impacted statements (F and F') in these two products (Equation 3.2).

$$invol_sim(s, p, p') = \frac{|\{B \cap B'\} \cup \{F \cap F'\}|}{|B \cup F \cup B' \cup F'|} \tag{3.2}$$

The *suspicious-involvement score* of statement s in p is the *maximum* of the similarity of the suspiciously-involving statement set of s in p to these sets of s in the failing products as shown in Equation 3.3. The reason for the use of the max function is that CLAP would like to consider the most similarly s behaves in p compared to the other failing products of the system. Intuitively, if p contains more suspiciously-involving statement sets which are more similar to such sets in the failing products, p is more likely to be *false-passing*.

$$sis(s, p) = \max_{p' \in P_F} invol_sim(s, p, p') \tag{3.3}$$

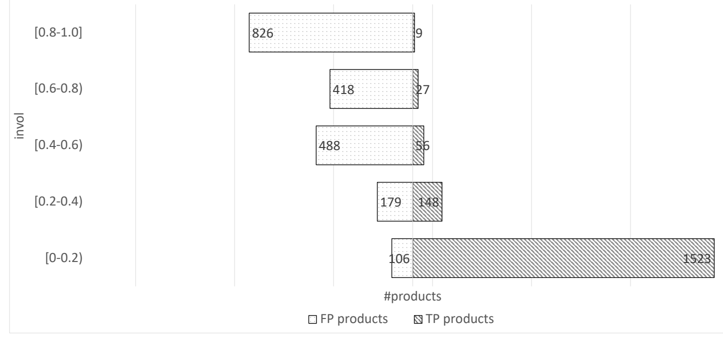


Figure 3.3: The presence of bug-involving statements in the passing products

Equation 3.4 shows how the possibility that p contains bug-involving statements is measured. This bug-involving statement containing possibility of p is aggregated from the suspicious-involvement scores of all suspicious statements.

$$invol(p) = \sum_{s_i \in \mathcal{S}} sis(s_i, p) \tag{3.4}$$

Figure 3.3 shows the suspicious-involvement scores of the passing products in the *verification dataset*. As seen, 85% of the *true-passing* products have scores less than 0.2. Meanwhile, about 90% of the *false-passing* products have scores greater than 0.2. Specially, the scores of about 40% of *false-passing* products are in the top range, i.e., [0.8-1.0]. It shows that in the *false-passing* products, suspicious statements frequently impact/be impacted by the other statements in the similar way they do in the failing products.

3.3.2 Test Adequacy

In general, assuring the quality of a program requires an adequate test suite which can cover a large number of the program's elements such as statements, branches, or paths [45]. If a program is tested by an inadequate test suite, the state of passing all the cases in the test suite could not be reliable, since a large portion of the program's elements is not tested (thoroughly). Although adequacy criteria are all imperfect, they are useful indication for determining the inadequacy of test suites [45]. For simplicity, CLAP measures the adequacy of a test suite in statement coverage. Additionally, to evaluate the fault diagnosability of test suites, CLAP also applies DDU [2] [98], which is a simple and effective criterion, as an adequacy attribute of the test suite.

[2] *DDU* is an acronym for *Density-Diversity-Uniqueness*.

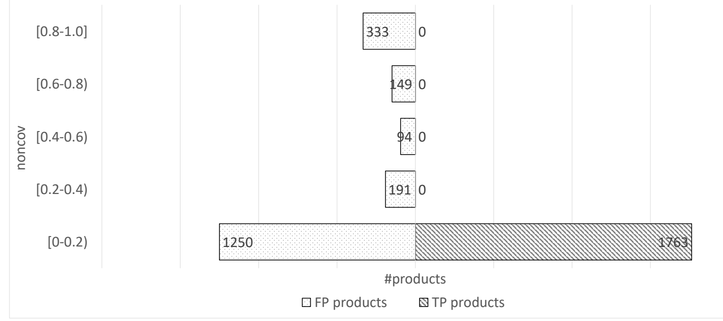


Figure 3.4: The portion of suspicious statements in the passing products which are not covered by their test suites

Code coverage

The prerequisite condition for a bug to be revealed in a product is that the buggy statement is reached (executed) by a test. Indeed, an adequate test suite should widely cover the suspicious statements contained in the product. *The more suspicious statements which are* ***not covered*** *by the test suite of the product, the less reliable the overall passing state of the product is.*

In general, for the given SPL system, any statement executed by any failed test in a failing product is suspicious to be a buggy one. Let $\mathcal{S} = \{s_1, ..., s_k\}$ be the set of suspicious statements, and S_p be the statements of a passing product $p \in P_P$. The set of suspicious statements in p is $K = \mathcal{S} \cap S_p$. Specially, K is categorized into two sets, K_e and K_{ne}, such that $K = K_e \cup K_{ne}$ where K_e and K_{ne} are the sets of statements which are *covered* and *not covered* by the test suite T of p. The *noncoverage rate* of p which reflects the portion of suspicious statements in p, yet not covered by T, is measured by the Equation 3.5.

$$noncov(p, T) = \frac{|K_{ne}|}{|K|} \tag{3.5}$$

Figure 3.4 shows this *code coverage* attribute of the passing products in the *verification dataset*. In fact, the general statement coverage of the test suites of the experimental data is quite high. This means that almost the statements in each product are covered by the test suite, and very few number statements in the product are not covered. That is the reason why in Figure 3.4 most of the products (both the *true-passing* and *false-passing* products) have small portions of non-coverage. However, this attribute is still useful to detect *false-passing* products, since various *false-passing* products have higher noncoverage rates than the others. As seen, 100% of the *true-passing* products have the noncover-

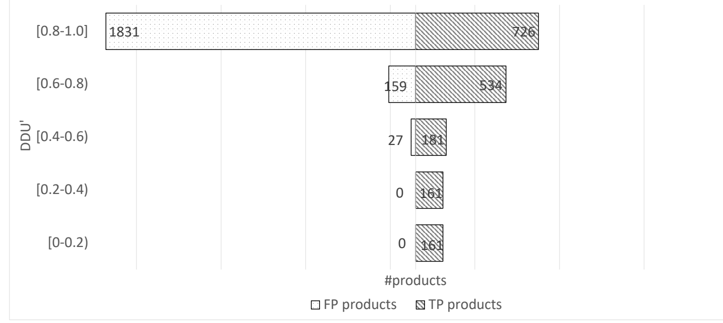


Figure 3.5: The undiagnosability (DDU') of the passing products' test suites

age rates lower than 0.2, while the noncoverage rates of 40% *false-passing* products are greater than this number. This result shows that the test suites of the *false-passing* products usually do not cover their suspicious statements. Meanwhile, the test suites of the *true-passing* products often have better coverage of the suspicious statements.

Density-Diversity-Uniqueness

In addition, to thoroughly evaluate the test suite, CLAP takes into account *DDU* metric proposed by Perez et al. [98] to evaluate the adequacy of the test suite regarding *fault diagnosability*. While the *coverage attribute* abstracts the execution information of test executions to favor an overall assessment of the suite, *DDU* takes into account per-test execution information, so it provides further insight about each test case of the suite.

The main idea of *DDU* is that a high-quality test suite must contain the test cases such that program elements are frequently tested (*density*) in diverse combinations (*diversity*), as well as the corresponding execution vectors of the elements in the program spectrum are distinguishable (*uniqueness*) [98]. The *DDU* value is from 0.0 to 1.0, and the *DDU* of an ideal test suite of a product is 1.0. Thus, the product whose test suite with a lower *fault diagnosability* in *DDU* is more likely to be *false-passing*. Specially, for a product p, the "undiagnosablity" of its test suite is $DDU'(p, T) = 1 - DDU(p, T)$ where $DDU(p, T)$ is the DDU value of the test suite T of product p. As a result, the higher $DDU'(p, T)$, the lower-quality test suite, and p is more likely to be a *false-passing* product.

Figure 3.5 shows DDU' of the passing products in the *verification dataset*. There are about 90% of the *false-passing* products have *DDU'* in the range of $[0.8, 1]$, while 60% of the *true-passing* products have *DDU'* in the lower range. This shows that the *true-passing* products have more diagnosable test suites, so their states of passing all the tests are more

reliable. With the less diagnosable test suites, the overall test results of the *false-passing* products are less reliable.

In practice, there are multiple criteria to evaluate the adequacy of the test suites, such as branch coverage, path coverage, mutation score, etc. These criteria could be applied using the same principle. Although sophisticated criteria could provide a more comprehensive adequacy evaluation, the computation could be expensive. For example, mutation score is a popular and powerful metric to evaluate the test suite. However, to calculate the mutation score, a large number of mutants need to be tested against the original test suite. This could be costly and time-consuming, especially for the large and complex systems containing many sampled products with large test suites [134–136]. Thus, to ensure the efficiency of the proposed attributes measuring the test suite's adequacy, CLAP employs statement coverage and DDU.

3.3.3 Test Effectiveness

For a product p, the fault-detecting effectiveness of its test suite T shows how intensively T tests the product's behaviors and T's ability to explore the product's incorrectness [46, 47]. To evaluate the effectiveness of the suite T, CLAP aims to investigate T by two attributes: *incorrectness verifiability* and *correctness reflectability.* For a passing product p, *incorrectness verifiability* measures how p's test suite, T, covers the product's suspicious behaviors. Meanwhile, *correctness reflectability* indicates that the passed tests of the product really reflect its correct behaviors (i.e., these tests are not just coincidentally passed).

Note that, the test adequacy attributes focus on how and to what extent the product's elements are covered by the test suite. These attributes do not take into account the results of the test cases. Meanwhile, the test effectiveness attributes focus on how the suite verifies the product's behaviors. Not only the product's elements but also the results of the tests are considered in the evaluation process.

In practice, the behaviors of the products are dynamically considered regarding the execution vectors in the product's spectrum. *The vector for a failed test represents an incorrect behavior of the product*, so-called *incorrect behavior vector.* In a buggy system, the incorrect behaviors are represented by the incorrect behavior vectors in the failing products. These vectors record the executions of the failing products in the failed tests. Meanwhile, the *correct behaviors* are represented by executions of truly passed tests which are not just

coincidentally passed. Thus, this work leverages execution vectors in spectra to measure *incorrectness verifiability* and *correctness reflectability.*

Incorrectness verifiability

From the incorrect behaviors of the failing products P_F of the given system $\mathfrak{S}$, CLAP identifies the behaviors which are *suspiciously* contained in a passing product $p \in P_p$ and then evaluate whether these behaviors are covered by the test cases of p. Intuitively, if p contains suspicious behaviors, these behaviors should be verified by test cases in its test suite, T, to sufficiently ensure the correctness of the product. To evaluate the reliability of the state of being a passing product of p with its test suite T, CLAP *measures the number of incorrect behaviors are (i) potentially contained in* p *and (ii) not covered by the test cases in* T. Intuitively, the correctness of p is less reliable if p potentially contains more such incorrect behaviors.

For (i), an incorrect behavior expressed by the execution vector v is *potentially contained* in p if p contains a large portion of the executed statements in v. This portion should be larger than a threshold $\mathcal{R}_{I1}$. The reason is that the products of an SPL system are composed of different sets of features, a product rarely contains a set of statements which is exactly the same as an execution of a test in another.

For (ii), the behavior expressed by v is *not covered* by the test in p if v is not similar to any execution vector in p's spectrum. Similar to (1), two execution vectors are similar if they share a large portion executed statements and this portion should be greater than a threshold $\mathcal{R}_{I2}$.

For (i), let $\mathbf{V}_{IB} = \{v_{f1}, ..., v_{fn}\}$ be the set of all the incorrect behavior vectors of the failing products P_F. Also, let S_p be the set of statements of the passing product p, and $V_p = \{v_1, ..., v_m\}$ be the set of the execution vectors in p's spectrum. Product p *potentially contains* a behavior presented by an incorrect behavior vector, $v_{fi} \in \mathbf{V}_{IB}$, if S_p contains a large portion of the statements executed in that vector. Equation 3.6 shows how CLAP identifies whether incorrect behavior v_{fi} is potentially contained by product p.

$$\frac{|\{s \in v_{fi} | v_{fi}[s] = 1 \land s \in S_p\}|}{|\{s \in v_{fi} | v_{fi}[s] = 1\}|} \geq \mathcal{R}_{I1} \tag{3.6}$$

For (ii), the behavior expressed by v_{fi} is covered by the test cases in T if there exists an execution vector recorded in p's spectrum similar to v_{fi}. To calculate the similarity of two

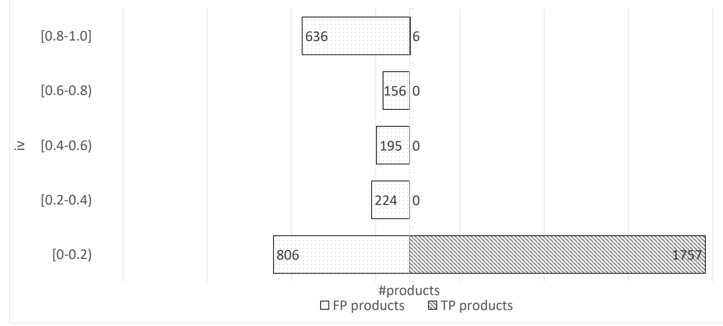


Figure 3.6: The incorrectness verification of the passing products' test suites

vectors, Jaccard is adopt as a similarity formulation. An execution vector $v_j \in V_p$ is similar to v_{fi} if the condition in Equation 3.7 is satisfied. In this Equation, a is the number of statements that executed in both v_j and v_{fi}, $a = |\{s \in v_j | v_j[s] = 1 \wedge v_{fi}[s] = 1\}|$. b is the number of statements executed in v_j but not in v_{fi}, $b = |\{s \in v_j | v_j[s] = 1 \wedge v_{fi}[s] = 0\}|$. Meanwhile, c is the number of statements which are not executed in v_j but executed in v_{fi}, $c = |\{s \in v_j | v_j[s] = 0 \wedge v_{fi}[s] = 1\}|$.

$$sim(v_j, v_{fi}) = \frac{a}{a+b+c} \geq \mathcal{R}_{I2} \tag{3.7}$$

The incorrectness verifiability score of the product p is calculated as the portion of the incorrect behaviors potentially contained by p but not covered by any test in the product's test suite T. Let I_1 be the set of incorrect behaviors vectors whose executed statements are contained in p. Also, let I_2 be the set of incorrect behaviors vectors in I_1 and not similar to any execution vector in the spectrum of p. The incorrectness verifiability attribute is measured as Equation 3.8.

$$iv(p, T) = \frac{|I_2|}{|I_1|} \tag{3.8}$$

Figure 3.6 shows the portion of the incorrect behaviors suspiciously contained in the passing products in the *verification dataset*, yet not tested by the products' test suites. In most *true-passing* products, the suspicious behaviors are covered by at least a test in their suites. However, two-thirds of the *false-passing* products contain suspicious behaviors, and these behaviors are not tested by the products' tests. This shows that the *false-passing* products' test suites are often ineffective in verifying their suspicious behaviors.

Correctness reflectability

In practice, not all of the passed tests can reliably confirm the success of the program since they could be coincidentally correct. Also, coincidentally passed tests are unbefitting for verifying the correctness of the product [21]. To determine if the correctness of a product is reliably reflected by its passed tests, CLAP measures the portion of passed tests of p which are likely to be *truly* correct. The more truly passed tests, the more effective p's test suite. Intuitively, with fewer truly passed tests, p has a higher possibility to be *false-passing*.

For a passing product p, *a passed test in p can truly represent a correct behavior if its execution vector is similar to any truly passed test's execution vector of any failing product* of the system. In a failing product, a test whose execution vector *is not similar* to the execution vector of a failed test can be considered as a truly passed test. This is because these tests are less likely to execute the faults revealed in the failed tests. Meanwhile, a passed test whose execution vectors *are similar* to the execution vector of a failed test, could be coincidentally passed. The reason is that with the similar execution vector but only one of them can reveal the bugs, then the other could reach the buggy statements but is ineffective in exposing the bugginess.

Given an SPL system $\mathfrak{S}$, let $\mathbf{V}_{IB} = \{v_{f1}, ..., v_{fn}\}$ and $\mathbf{V}_{CB} = \{v_{p1}, ..., v_{pm}\}$ be the sets of incorrect and correct behaviors of the failing products. In other words, $\mathbf{V}_{IB}$ and $\mathbf{V}_{CB}$ are sets of execution vectors of the failed tests and passed tests in the failing products of system $\mathfrak{S}$. A passed test in a failing product whose execution vector $v_{pi} \in \mathbf{V}_{CB}$, is truly passed if its execution vector is not similar to any failed test's execution vector, formally, $\nexists v_{fj} \in \mathbf{V}_{IB}, sim(v_{pi}, v_{fj}) \geq \mathcal{R}_C$, where $\mathcal{R}_C$ is a similarity threshold.

Let $V_p = \{v_1, ..., v_k\}$ be the set of execution vectors of passing product p with its test suite T. The correctness reflected by a passed test in T, whose execution vector $v_i \in V_p$, is confirmed if v_i is similar to a truly correct test's execution vector in the failing products. Specifically, $\exists v_{pj} \in \mathbf{V}_{CB},\ \nexists v_{ft} \in \mathbf{V}_{IB}, sim(v_{pj}, v_{ft}) \geq \mathcal{R}_C$, and $sim(v_{pj}, v_i) \geq \mathcal{R}_C$.

Let C be the number of truly correct tests of p, the correctness reflectability value of a passing product p is measured by Equation 3.9. The correctness reflectability value is 0 when all the passed tests of the passing product p are confirmed by truly correct tests of the failing products. This value is 1 when none of the tests reflecting the correctness of p is confirmed.

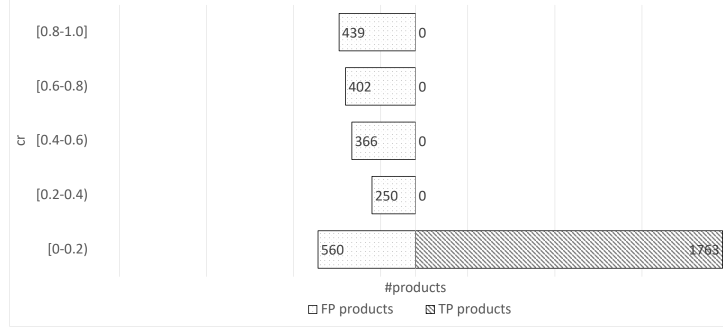


Figure 3.7: The correctness reflectability of the passing products' test suites

$$cr(p, T) = 1 - \frac{C}{|V_p|} \tag{3.9}$$

Figure 3.7 shows the *correctness reflectability* of the passing products in the *verification dataset*. The figures of this attribute of all the *true-passing* products are in the lowest range, i.e., [0, 0.2). This means that the correctness reflected by almost passed tests in the *true-passing* products is confirmed by at least a truly passed test of a failing product. Meanwhile, more than 70% of the *false-passing* products contain passed tests, but the correctness reflected by their passed tests could not be confirmed. This indicates the *false-passing* products often contain passed tests, yet many of them could not represent the products' true correctness.

3.3.4 Detecting *False-passing* Products

This work considers the problem of *false-passing* product detection as a binary classification problem. Specifically, the possibility that a product p is *false-passing* is measured by the proposed attributes regarding product's implementation and its test quality. Product p is represented by a six-dimension vector $x = \langle a_1, a_2, a_3, a_4, a_5, a_6 \rangle$, where $a_1 = bscp(p)$ and $a_2 = invol(p)$ reflect whether the product's implementation contains buggy statements and the corresponding bug-involving statements. The remaining attributes reflect its test quality, $a_3 = noncov(p, T)$ and $a_4 = DDU'(p, T)$ are about test adequacy, while $a_5 = iv(p, T)$, and $a_6 = cr(p, T)$ are about test effectiveness. For each attribute, the higher score, the more likely p is *false-passing*.

A classifier $h(x)$, Equation 3.10, could be applied to predict the possibility that product presented by vector x is a *false-passing* product. The label y of the product is *false-passing* if the result of function $h(x)$ is greater than a threshold $\mathcal{R}_{fp}$, otherwise, it is a

true-passing product. In this work, the default threshold, $\mathcal{R}_{fp} = 0.5$, is used. However, one can further consider Precision-Recall curve to trade-off between precision and recall and select the optimal decision threshold.

$$y = \begin{cases} \textit{false-passing} & \text{if } h(x) \geq \mathcal{R}_{fp} \\ \textit{true-passing} & \textit{otherwise} \end{cases} \tag{3.10}$$

In practice, $h(x)$ could be any machine learning classifier such as Support Vector Machine, K-Nearest Neighbor, or Logistic Regression. For example, if Logistic Regression is selected as a classification algorithm, the classification formula of $h(x)$ is $h(x) = \frac{1}{1+e^{-(wx+b)}}$, in which w and b are weights and a bias term that learned via training. The impact of different classifiers on the performance of CLAP is empirically evaluated in Section 3.6.

3.4 Mitigation of Negative Impact of *False-passing* Products on Variability Fault Localization

As the negative impact on variability FL is caused by *false-passing* products' unreliable test results, there are two main directions to mitigate their negative impacts. Specifically, either the reliability can be improved or the unreliability can be eliminated in the test results of these products. Firstly, the reliability of test results can be improved by enhancing the quality of the test suites of the *false-passing* products. Secondly, the unreliability of test results can be eliminated by removing products whose unreliable test results and/or removing low-quality test cases.

First, *the FL performance can be boosted by improving the reliability of the false-passing products' test results.* Specifically, the *false-passing* products can be more thoroughly tested by a better test suite to explore their bugginess. Once their bugs are revealed by test cases, CLAP not only has more information about the faults but also has better assessments of the (overall) test results of the products. Intuitively, this could help improve the performance of variability FL.

In particular, the failure indications (e.g., suspicious statements and behaviors) can be used to guide test generators to produce better test suites for these *false-passing* products. Specially, the newly generated test cases should focus on these failure indications. For instance, to have a more adequate test suite, new test cases can be added to cover the suspicious statements which have not been covered yet (Section 3.3.2). To improve the

test effectiveness, new test cases could be added to verify whether the suspicious behaviors can cause failures for the *false-passing* products (Section 3.3.3).

Second, *the results of FL techniques can also be enhanced by eliminating the unreliability in the test results of the false-passing products.* Particularly, the unreliable test results can be at either the product-level or the test case-level. For the product-level, the test result is the overall test result (i.e., being failing or passing) of a product. Meanwhile, the test result at the test case-level is the result (i.e., failed or passed) of a single test case.

For the *product-level*, CLAP can remove all the detected *false-passing* products before localizing variability faults. This strategy reduces the number of buggy products, yet incorrectly considered as passing ones. Thus, it can benefit the *product-base assessment* FL techniques [6] which evaluate the suspiciousness of the statements according to the number of failing and/or passing products.

For the *test case-level*, the coincidentally passed tests should be removed. The reason is that the coincidentally passed tests improperly increase the number of passed tests executed by the buggy statements [21]. As a result, these test cases can negatively impact the performance of *test case-based assessment* FL approaches [25] which evaluate the statements' suspiciousness based on the number of failed and/or passed tests. To clean such tests, each passed test of a product should be carefully investigated.

In summary, this section discusses the directions to mitigate *false-passing* products' impact on FL techniques. This work evaluates the effectiveness of two methods: adding new tests to improve the product's test quality and eliminating the unreliability at the product-level. The mitigation direction eliminating coincidentally passed tests required thorough investigation to carefully review and eliminate unreliable tests. The techniques in this direction are beyond the scope of this work and left for future work.

3.5 Empirical Methodology

3.5.1 Research Questions

To evaluate CLAP in detecting *false-passing* products and mitigating their negative impacts on fault localization, this chapter seeks to answer the following research questions:

RQ1: Accuracy Analysis: How accurate is CLAP in detecting *false-passing* products?

RQ2: Mitigating Impact of *False-passing* Products on Fault Localization: How

Table 3.2: Products' test suites before and after being transformed

System	#Transformed products	Original test suite T		New test suite T'	
		#Tests	Cov (%)	#Tests	Cov (%)
BankAccountTP	1,833	22	96.7	19	92.8
Elevator	161	165	76.5	120	69.4
Email	420	94	97.2	89	96.0
ExamDB	126	136	96.6	128	96.4
GPL	6,091	90	98.7	88	98.2
ZipMe	541	253	42.9	249	42.7

#Tests and Cov stand for the number of tests and statement coverage in a product.

does CLAP mitigate the negative impact of *false-passing* products on the performance of state-of-the-art FL techniques including VARCOP and SBFL [29, 30]?

RQ3. Sensitivity Analysis: How does CLAP perform on different evaluation scenarios? And how do different training data sizes impact CLAP's performance?

RQ4. Intrinsic Analysis: How do the different attributes of CLAP contribute to the *false-passing* product detection performance?

RQ5. Time Complexity: What is CLAP's running time?

3.5.2 Dataset

To evaluate CLAP, *failing*, *true-passing*, and *false-passing* products in existing buggy SPL systems are systematically collected as follows.

First, to practically collect *false-passing* products, a random number of failing products in a buggy SPL system are transformed into passing products by removing all their failed tests. Indeed, for a failing product p with the original test suite T, removing all the failed tests in T creates a new test suite T'. Product p with the test suite T' is *false-passing* since p fails with T, but not with T'. The average number of tests in the original test suite T and the new test suite T', as well as the corresponding statement coverage of the products in each system are shown in Table 3.2.

Although removing failed tests in T slightly affects the statement coverage of p (i.e., about 2%), *this data collecting procedure is independent of the proposed approach.* By this

procedure, only failed tests in T are removed, and all the passed tests in the original test suite of p are kept unchanged. Meanwhile, the attributes of CLAP measure how likely p is a *false-passing* product by investigating its implementations and its passed tests.

Second, to collect *true-passing* products, the overall test results of the passing products are verified and labeled accordingly. In practice, this process could be very tedious and time-consuming even for experts. To practically verify the overall test result of a passing product p', this work proposes a semi-automated procedure generating tests and making the product fail in three steps. If p' fails any test in each step, then p' with its original test suite is a *false-passing* one. Otherwise, it can be considered as a *true-passing* product.

Step 1. Automatically generating new test cases for p'. Multiple test generation tools, e.g., Evosuite [137] and Randoop [138], are used to generate new tests for p'. If the product is failed by any added test, the product with the original test suite is a *false-passing* product. If the product still passes all the added tests, moving to the next step.

Step 2. Applying a hybrid test generator. The bugs of the system have been explored by failed tests of the failing products. Thus, these failed tests can be used as the guidance to verify the existence of the bugs in p'. Specifically, failed tests of the failing products are tried to adapt to test p'. In addition, for an SPL system, each product is composed from different set of features, thus tests need to be adapted appropriately according to each product, and not all of the failed tests can be adapted to test another products. If any adapted test can be executed by p' and p' creates incorrect output, p' with the original test suite is a *false-passing* product. Otherwise, moving to the next step.

Step 3. Manually investigating the product. p' is manually investigated and generated tests to carefully confirm its correctness or bugginess. If p' fails any newly generated test, p' with the original test suite is *false-passing*. Otherwise, if the product still passes all the manually generated tests, p' with the original test suite is a *true-passing* product.

This work applied the procedure on the sampled products of the buggy systems in the dataset collected by Ngo et al. [5]. This dataset includes 1,570 buggy versions with their sampled products and the corresponding test suites of six Java SPL systems which are widely used in SPL studies. A *buggy version* is a version of the SPL system which contain one or more bug. In other word, a buggy version can be considered as a buggy system. After labeling, there are 823 buggy versions which contains large numbers of products in all three kinds: failing, *true-passing*, and *false-passing* products. All the other buggy versions which are not satisfied are removed. Table 3.3 shows the overview of the dataset.

Table 3.3: Dataset overview

System	Buggy versions			Products		
	1-Bug	2-Bug	3-Bug	#Fs	#FPs	#TPs
BankAccountTP	41	117	29	2,055	2,328	1,975
Elevator	14	17	10	217	326	195
Email	14	21	34	553	587	723
ExamDB	10	44	23	201	127	288
GPL	97	188	70	6,612	9,995	18,538
ZipMe	17	46	31	686	828	836
Total				10,433	14,191	22,555

#Tests and Cov stand for the number of tests and statement coverage in a product.
N-Bug represents the number of bugs (i.e., N) in the buggy version.
#Fs, #FPs, and #TPs stand for failing, *false-passing*, and *true-passing* products.

3.5.3 Empirical Procedure

Accuracy analysis. 823 buggy versions are split into five folds (5-fold cross-validation). Specifically, four folds are picked for training and one remaining fold is used for testing. This work adapted cross-validation to aggregate average results for the final assessment.

Mitigating impact of *false-passing* products on fault localization. The buggy versions in five systems are used to train the *false-passing* product detection model, and then the trained model was used to detect *false-passing* products in the buggy versions of the remaining system. Next, the effectiveness of the proposed mitigation techniques on FL performance is evaluated. The *original* FL performance is measured using all the sampled products and their corresponding testing information. The performance after applying the mitigation techniques: (1) removing all the detected *false-passing* products, only using the sampled products which are not predicted to be *false-passing* and their testing information (*removing FPs*); (2) generating new tests for detected *false-passing* products for further testing, then if the faults are revealed (i.e., at least one added tests is failed), these products are used with the other products which are not predicted to be *false-passing* to localize faults (*adding tests*).

Sensitivity analysis. This work conducted two experiments to measure the impact of different evaluation scenarios and training data sizes.

First, this dissertation posits that *different systems, buggy versions of a system, and products in a version have different degrees of relevance.* Hence, to evaluate the impact of

the specialities of systems, buggy versions, and products on CLAP, the following scenarios are experimented.

- **System-based edition**. CLAP is trained with the products in buggy versions of five systems, and the products in the buggy versions of the remaining system are used for testing. In practice, this setting reflects the case when the development history of the system is not very long, and the data about the current system is not available/sufficient. Thus, the data of the other systems are leveraged for detecting *false-passing* products in the developing system.

- **Version-based edition**. All the buggy versions of the six systems are shuffled and then split into a training set and a testing set by the ratio 8:2. This scenario comes from the idea when the system is developed for a while, and the information from other systems, as well as the other buggy versions of the developing system are available for detecting new *false-passing* products.

- **Product-based edition**. All the products in all the buggy versions of the six systems are shuffled and then split into a training set and a testing set by the ratio 8:2. The idea of this scenario is that during assuring systems' quality, the sampled products are progressively tested and determined if they are *false-passing*. Thus, the detected and confirmed *false-passing* products could be used for training and detecting other *false-passing* products.

- **Within-system edition**. The experiment on the buggy versions of each system is conducted. The buggy versions of a system are split into a training set and a testing set by the ratio 8:2. This is also another real-world setting. When the data from other systems is not available, then only the information from testing system is used to detect *false-passing* in the system.

Second, as CLAP is data-driven, this experiment studies the **impact of training data sizes** on CLAP's performance. A system is randomly picked for testing. The training data is gradually increased by adding data from the remaining systems.

Intrinsic analysis. To better understand the proposed approach, this experiment studies the impact of the attributes on CLAP's performance: product implementation, test adequacy, and test effectiveness. Different variants of CLAP are built, each variant use

attributes only in one of these aspects to detect *false-passing* products, and measure their performance.

3.5.4 Metrics

This work adopt *Accuracy*, *Precision*, *Recall*, and *F1-score* which are widely used to evaluate classification model [139]. Let P_{TP} and P_{FP} be the sets of predicted *true-passing* and *false-passing* products, and A_{TP} and A_{FP} be the sets of actual *true-passing* and *false-passing* products, respectively. These metrics are measured as the following formulas.

Acc. Accuracy measures the general performance of the classification model. Accuracy indicates the ratio of correctly predicted *true-passing* and *false-passing* products out of the total passing products. Equation 3.11 shows how to measure Accuracy of the models.

$$Acc = \frac{|\{P_{TP} \cap A_{TP}\} \cup \{P_{FP} \cap A_{FP}\}|}{|A_{TP} \cup A_{FP}|} \tag{3.11}$$

Prec. Precision indicates the prediction correctness for *true-passing*, $Prec(TP)$, or *false-passing*, $Prec(FP)$. The Precision for *true-passing* and *false-passing* product prediction are measured by Equation 3.12 and Equation 3.13, respectively.

$$Prec(TP) = \frac{|P_{TP} \cap A_{TP}|}{|P_{TP}|} \tag{3.12}$$

$$Prec(FP) = \frac{|P_{FP} \cap A_{FP}|}{|P_{FP}|} \tag{3.13}$$

Recall. Recall indicates the effectiveness of prediction, the portion of correctly predicted *true-passing* (*false-passing*) products out of the total number of actual *true-passing* (*false-passing*) products. Recall of *true-passing* and *false-passing* product detection are measured by Equation 3.14 and Equation 3.15, respectively.

$$Recall(TP) = \frac{|P_{TP} \cap A_{TP}|}{|A_{TP}|} \tag{3.14}$$

$$Recall(FP) = \frac{|P_{FP} \cap A_{FP}|}{|A_{FP}|} \tag{3.15}$$

F1-score. F1-score is defined as the harmonic mean of precision and recall, it indicates balance between those. F1-scores of *true-passing* and *false-passing* product detection are measured by Equation 3.16 and Equation 3.17, respectively.

$$F1(TP) = 2 * \frac{Prec(TP) \times Recall(TP)}{Prec(TP) + Recall(TP)} \tag{3.16}$$

$$F1(FP) = 2 * \frac{Prec(FP) \times Recall(FP)}{Prec(FP) + Recall(FP)} \tag{3.17}$$

To measure FL performance, *Rank* and *EXAM* are employed. These metrics are widely used in the existing FL studies [4, 25].

Rank. Rank indicates the position of the buggy statements in the resulted lists of the FL techniques. The lower rank of buggy statements, the more effective approach. If there are multiple statements having the same score, buggy statements are ranked last among them. For the cases of multiple bugs, this work measured *Rank* of the first buggy statement (*best rank*) in the resulted lists.

EXAM. EXAM [140] is the proportion of the statements needs to be examined until the first faulty one is reached (Equation 3.18). In Equation 3.18, r is the rank of the buggy statement and N is the total number of statements in the list. The lower *EXAM*, the better FL technique.

$$EXAM = \frac{r}{N} \times 100\% \tag{3.18}$$

3.5.5 Experimental Setup

Classifier selection. Six popular classifiers are selected based on their use in the related studies [141, 142]. The classifiers include Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Naïve Bayes, Logistic Regression, Decision Tree, and Random Forest.

Experimental setup. The classification models are implemented using *Sklearn*. For each classifier, this work trains the model with the respective standard settings. Each item in the dataset is a passing product represented by a 6-dimensional vector whose values are computed based on six attributes proposed in Section 3.3. The training and the testing sets in each experiment are separated by different scenarios as described in

Table 3.4: Accuracy of *false-passing* product detection model

Classifier	Product	Precision	Recall	F1-Score	Accuracy
SVM	True-passing	88.16%	**97.09%**	92.41%	90.04%
	False-passing	**94.19%**	78.36%	85.55%	
KNN	True-passing	90.41%	93.97%	92.16%	90.02%
	False-passing	89.30%	83.46%	86.28%	
Naïve Bayes	True-passing	88.36%	95.25%	91.68%	89.21%
	False-passing	90.95%	79.18%	84.66%	
Logistic Regression	True-passing	88.75%	95.99%	92.23%	89.91%
	False-passing	92.30%	79.81%	85.60%	
Decision Tree	True-passing	90.03%	96.26%	93.04%	91.01%
	False-passing	92.99%	82.30%	87.32%	
Random Forest	True-passing	89.81%	95.00%	92.33%	90.16%
	False-passing	90.83%	82.12%	86.26%	

Section. 3.5.3. The experiments are conducted on a desktop with Intel Core i5 2.7GHz, 8GB RAM.

3.6 Experimental Results

3.6.1 Accuracy Analysis (RQ1)

As shown in Table 3.4, CLAP *is highly effective in detecting false-passing products for all the studied classifiers.* The average accuracy for all six classifiers is about 90%. This figure indicates that CLAP can correctly detect 9 out of 10 products to be *true-passing* or *false-passing*. The average F1-scores of *true-passing* and *false-passing* product detection are also high, about 92% and 86%, respectively. Furthermore, the average *Recall* for *false-passing* product detection is about 81%. In other words, if there are 10 *false-passing* products, 8 of them are correctly detected. Meanwhile, this figure for *true-passing* product classification is approximately 96%, which demonstrates that almost *true-passing* products can be accurately detected.

Among the studied classifiers, Decision Tree has the best *Accuracy* (91.01%). This indi-

Table 3.5: Mitigating the *false-passing* products' negative impact on FL performance

Metric	**Ranking formula**	**VarCop**			**SBFL**		
		Original	Removing FPs	Adding tests for FPs	Original	Removing FPs	Adding tests for FPs
Rank	Tarantula	3.35	2.52	2.22	5.10	4.75	4.53
	Ochiai	2.39	2.23	2.28	3.00	2.77	2.86
	Op2	4.31	4.18	4.33	7.03	6.84	6.96
	Barinel	3.69	2.83	2.91	5.10	4.74	4.53
	Dstar	2.55	2.14	2.19	3.06	2.91	2.98
EXAM	Tarantula	1.35	1.10	1.00	1.89	1.86	1.82
	Ochiai	1.02	1.01	0.97	1.12	1.09	1.11
	Op2	1.40	1.38	1.40	2.29	2.24	2.27
	Barinel	1.46	1.21	1.22	1.89	1.86	1.82
	Dstar	1.01	0.94	0.90	1.14	1.09	1.12

cates that Decision Tree most effectively separates *true-passing* and *false-passing* products in a set of passing products. Meanwhile, SVM obtains the highest *false-passing* product detection *Precision* and *true-passing* product classification *Recall*. This demonstrates that compared to the other classifiers, although SVM can detect fewer numbers of *false-passing* products (the SVM's *Recall* on *false-passing* product detection is slightly lower than the other classifiers' results), it more precisely predicted *false-passing* products (its *Precision* on *false-passing* product detection is higher than the others classifiers). At the same time, it also less erroneously detected *true-passing* products (higher *Recall* on *true-passing* product detection). In fact, misleadingly detecting *true-passing* products can cause missing useful information in FL procedure and it could negatively affect FL performance. Thus, for the studied dataset, SVM is the safest and most suitable for Clap, and SVM is used for the following experiments.

3.6.2 Mitigating Impact of *False-passing* Products on Fault Localization (RQ2)

Table 3.5 shows the performance of the state-of-the-art variability fault localization techniques in three settings, the original performance and the FL performance after applying the mitigation methods: removing *false-passing* products and adding tests for *false-passing* products. In this experiment, Clap used SVM to detect *false-passing* products.

As shown in Table 3.5, *removing "noises" caused by false-passing products helps both* VarCop *and SBFL obtain better performance compared to when they are applied on the original testing information.* Specifically, when *false-passing* products are detected and removed, the performance of the VarCop improved up to 25% in Rank and 19% in EXAM, also these improvements of SBFL are 8% and 3%, respectively. Indeed, the presence of *false-passing* products could lower the suspicious scores of the incorrect statements. The reason is that these products not only decrease the ratio of failing and passing products containing the buggy statements but also decrease the ratio of failed tests and passed tests executed by these statements. This causes confusion for the FL techniques which often distinguish the incorrect statements from the others based on the number of failing/passing products, as well as the number of failed/passed test cases. Therefore, eliminating *false-passing* products can help to improve the performance of FL techniques.

Listing 3.1: A variability bug in the feature DailyLimit of system BankAccountTP

```
1  boolean update(int x){
2      int newWithdraw = withdraw;
3      if (x < 0){
4          newWithdraw += x--;
5          //Patch:  x-- => x
6          if (newWithdraw < DAILY_LIMIT) {
7              return false;
8          }
9      }
10     //...
11 }
```

Listing 3.1 shows a variability bug (line 4) in the BankAccountTP. The system is sampled into 34 products for testing. After testing, the bug is revealed in 2 products, i.e., there are 2 failing products and 32 passing products. By using the program spectra of all of these 34 products for localizing fault, VarCop ranks the buggy statement at 7th and SBFL ranks it at 5th. However, among the passing products, there are 14 *false-passing* products. *After detecting and removing false-passing products, FL performance of* VarCop *and SBFL is improved 30% and 40%, respectively.* Specifically, VarCop ranks the bug at 5th and SBFL ranks it at 3rd. In this case, Clap correctly predicted 13 products as *false-passing*. As a result, a large number of products, which contain the bug and are incorrectly considered as passing products, have been removed. Moreover, the results showed that by removing *false-passing* products, 52 coincidentally passed tests in these products are also removed. Therefore, the buggy statement becomes more distinguishable from the other statements of the system in terms of both product-based and test case-based assessment.

Additionally, *when the products predicted as false-passing are further tested by better quality test suites, FL techniques have more useful information to effectively detect the faults.* Specifically, when more tests are added for further testing *false-passing* products, the performance of VarCop is improved up to 34% in Rank and 26% in EXAM, also these figures of SBFL are 11% and 4% compared to that they are applied on the original testing information. In Listing 3.1, after adding new tests for detected *false-passing* products, the bug is revealed in 2 more products, increasing the number of failing products from 2 to 4. By using the spectra of these 4 failing products and 18 predicted *true-passing* products of the systems, both VarCop and SBFL could rank the bug at 3rd, which is also better than directly removing all 13 detected *false-passing* products.

Listing 3.2: A variability bug in the feature ExamDataBaseImpl of system ExamDB

```
1  public boolean consistent(){
2     for (int i = 0; i < students.length; i--) {
3     //Patch:  i-- => i++
4        if (students[i] != null && !students[i].backedOut && students[i].points < 0) {
5           return false;
6        }
7     }
8     return true;
9  }
```

However, *in some cases, the FL performance after adding new (failed) tests for false-passing products could be worse than that when removing all of the detected false-passing products.* The reason is that besides the added tests which are failed, the original test suites of the *false-passing* products already contain test cases which coincidentally passed (unreliable passed tests). Such tests also produce noises and negatively affect FL performance. In Listing 3.2, the bug (line 2) is ranked 9th, 3rd, and 5th by SBFL in the original setting and the two mitigation settings. By adding tests to the detected *false-passing* products, the buggy statement's rank is worse than that of removing all of these products (5th vs. 3rd). In this case, two *false-passing* products are detected and via added test cases, the bug is revealed in these products. However, in their test suites, there are 15 coincidentally passed tests which executed the faults but cannot reveal the failure. Consequently, using the program spectra of these products with the original test suites and added tests decreased the suspicious score of the buggy statement 0.01 and this causes its rank becomes worse. For these *false-passing* products, an analysis to remove the unreliable passed tests and add new effective test cases should be designed to help FL

Table 3.6: Impact of different experimental scenarios

Edition	Product	Precision	Recall	F1-Score	Accuracy
System-based	True-passing	85.51%	92.16%	89.15%	88.44%
	False-passing	89.42%	85.83%	86.83%	
Version-based	True-passing	88.16%	97.09%	92.41%	90.04%
	False-passing	94.19%	78.36%	85.55%	
Product-based	True-passing	87.53%	96.97%	92.01%	89.70%
	False-passing	94.27%	78.26%	85.52%	
Within-system	True-passing	88.73%	96.29%	92.21%	92.29%
	False-passing	96.12%	87.02%	91.16%	

techniques improve their performance, and this is left for future work.

3.6.3 Sensitivity Analysis (RQ3)

Impact of evaluation scenarios

Table 3.6 shows the *false-passing* product detection performance of CLAP with SVM in four scenarios by different degrees of relevance of the training and testing data: System-based, Version-based, Product-based, and Within-system editions. Overall, *the more relevant training and testing data, the better performance of the false-passing product detection model.* Specifically, in the *System-based edition*, the training and testing data are *the least relevant* since the testing set contains the systems which are totally different from the training set. In this edition, the average classification accuracy is 88.44%. Meanwhile, in the *Within-system edition*, the training and testing data are *the most relevant* since both the training and testing data contain products in the buggy versions of the same system. Thus, these products could share some similar characteristics about the programs and tests. Intuitively, CLAP can better capture these attributes and better detect *false-passing* products. Specially, the performance of CLAP in the *Within-system edition* is 92.29% in *Accuracy* and 96.12% in *false-passing* product prediction *Precision*, which are higher than those of *System-based edition* about 4% and 7%, respectively.

The detail performance of CLAP in each system in the *System-based edition* and *Within-system edition* are shown in Table 3.7 and Table 3.8, respectively. For the System-based

Table 3.7: Clap's performance on each system in system-based edition

System	Product	Precision	Recall	F1-Score	Accuracy
BankAccountTP	True-passing	89.43%	99.14%	94.04%	94.13%
	False-passing	99.17%	89.73%	94.21%	
Elevator	True-passing	60.25%	91.83%	72.76%	74.23%
	False-passing	92.86%	63.69%	75.56%	
Email	True-passing	91.19%	84.51%	87.72%	86.95%
	False-passing	82.50%	89.95%	86.06%	
ExamDB	True-passing	100.00%	89.58%	94.50%	92.77%
	False-passing	80.89%	100.00%	89.44%	
GPL	True-passing	87.47%	94.66%	90.92%	87.71%
	False-passing	88.29%	74.82%	81.00%	
ZipMe	True-passing	96.84%	91.51%	94.10%	94.23%
	False-passing	91.88%	96.98%	94.36%	

Table 3.8: Clap's performance on each system in within-system edition

System	Product	Precision	Recall	F1-Score	Accuracy
BankAccountTP	True-passing	96.28%	98.23%	97.25%	97.39%
	False-passing	98.40%	96.34%	97.36%	
Elevator	True-passing	72.22%	92.86%	81.25%	85.71%
	False-passing	95.83%	82.14%	88.46%	
Email	True-passing	90.58%	97.66%	93.99%	92.98%
	False-passing	96.67%	87.00%	91.58%	
ExamDB	True-passing	98.04%	100.00%	99.01%	98.70%
	False-passing	100.00%	96.30%	98.12%	
GPL	True-passing	86.74%	97.52%	91.81%	88.48%
	False-passing	94.00%	70.71%	80.54%	
ZipMe	True-passing	88.54%	91.45%	89.97%	90.46%
	False-passing	92.26%	89.60%	90.91%	

edition, Clap obtained the best performance in BankAccountTP and ZipMe systems. For the Within-system edition, Clap correctly detected most of *true-passing* and *false-passing* products in the buggy versions of BankAccountTP and ExamDB systems. Meanwhile, in both editions, the accuracy of Clap in the buggy versions of Elevator is the lowest. One of the reasons is the existence of flaky tests in their test suites, which negatively affect the model in both the training and testing phases. The impact of such tests on the training phase is discussed in Section. 3.6.3.

In addition, Clap obtained better *false-passing* product detection *Recall* in the System-based edition (86%) compared to the Version-based edition and Product-based edition (about 78%), although these two later settings have better accuracy. This demonstrates that for the System-based edition, a product has a higher probability to be predicted as a *false-passing* product. This is because for the *false-passing* products of some systems, their values in several attributes are significantly smaller than those figures in the products of the other systems. Consequently, once Clap learns from these small values, and then predicts new passing products, more products will be predicted as *false-passing* products. For example, the fault diagnosability (DDU') attribute, the average value of the *false-passing* products of system Email is 0.81, while this figure of system ExamDB is only 0.6. If the model learns from products of system ExamDB and then predicts products of system Email, many products of this system, including *true-passing* products, could be predicted as *false-passing* products. As a result, the System-based edition obtained higher *false-passing* product detection *Recall*, yet lower *true-passing* product classification *Precision* compared to the Version-based edition and the Product-based edition.

The proposed approach performed stably in the Version-based and Product-based editions. The reason is that the data separation methods in these two scenarios are quite similar. The only difference is that in the Product-based edition, the training and testing sets could contain the products in the same buggy versions. Thus, the detection performance of Clap in the Version-based and Product-based editions is quite similar.

Impact of training data sizes

Table 3.9 shows the performance of Clap by the training data size with SVM. In general, *the larger the training data set, the better performance of* Clap. If the training set contains the buggy versions of only one system, Clap's *Accuracy* is 82.60% and its *false-passing* product classification *Precision* is 74.9%. When the training data increases to

Table 3.9: Impact of different training data sizes (the number of systems)

#System	Product	Precision	Recall	F1-Score	Accuracy
1	True-passing	92.02%	81.88%	86.65%	82.60%
	False-passing	**74.90%**	93.19%	83.05%	
2	True-passing	96.82%	63.07%	76.38%	78.47%
	False-passing	**68.18%**	97.44%	80.23%	
3	True-passing	95.37%	76.90%	85.14%	85.19%
	False-passing	**77.03%**	95.40%	85.24%	
4	True-passing	90.18%	81.33%	85.53%	84.81%
	False-passing	**79.48%**	89.10%	84.02%	
5	True-passing	91.19%	84.51%	87.72%	86.95%
	False-passing	**82.50%**	89.95%	86.06%	

five systems, the figures of CLAP are improved by about 5% and 7%, respectively. This is reasonable because by learning from more data, CLAP can observe and recognize better *false-passing* and *true-passing* products.

However, *if added training data contains noises, it could decrease the performance of* CLAP. For example, the *Accuracy* of CLAP dropped from 82.60% to 78.47% when the training set is increased from one to two systems. This figure also slightly declined, about 0.4%, when the training set increased from three to four systems. The reason is that in the added systems, there are several "flaky" tests in their buggy versions. The results of these tests are inconsistent, sometimes they are passed and sometimes failed without any code changes [143]. For example, some tests invoked `random()` function to get a random number, and then the tests failed due to the numbers are generated differently in each run, not because of the bugs in the system. Due to their inconsistent test results, these tests might create noises for *false-passing* product detection tools. Consequently, the performance of CLAP in these cases are decreased.

3.6.4 Intrinsic Analysis (RQ4)

To study the impact of each attribute set on CLAP's performance, several variants of CLAP were built, each of them enables a single attribute set: *product implementation*, *test adequacy*, and *test effectiveness*. In this experiment, these variants of CLAP were

Table 3.10: Impact of attributes on Clap's performance

System	Attributes	Product	Precision	Recall	F1-Score	Accuracy
GPL	Product implementation	True-passing	84.69%	87.71%	86.17%	81.52%
		False-passing	74.80%	**69.69%**	72.15%	
	Test adequacy	True-passing	80.45%	99.74%	89.06%	83.92%
		False-passing	99.07%	53.69%	69.64%	
	Test effectiveness	True-passing	78.74%	96.59%	86.76%	80.64%
		False-passing	88.50%	50.18%	64.05%	
	All	True-passing	87.47%	94.66%	90.92%	87.71%
		False-passing	88.29%	74.82%	81.00%	
ZipMe	Product implementation	True-passing	86.09%	99.16%	92.16%	91.53%
		False-passing	99.00%	**83.82%**	90.78%	
	Test adequacy	True-passing	76.82%	93.54%	84.36%	82.57%
		False-passing	91.61%	71.50%	80.32%	
	Test effectiveness	True-passing	87.39%	48.92%	62.73%	70.79%
		False-passing	96.84%	91.51%	94.10%	
	All	True-passing	96.84%	91.51%	94.10%	94.23%
		False-passing	91.88%	96.98%	94.36%	

applied in setting cross-system edition of the two largest systems, GPL and ZipMe. Note in this experiment, Clap used SVM, the impact of different classifiers and different setting editions on Clap's performance has been shown in Section 3.6.1 and Section 3.6.3.

The product implementation attributes help Clap *achieve better performance compared to the test adequacy and test effectiveness attributes.* The reason is that the *product implementation* attributes directly provide information about buggy statements in the products. The model using these attributes can capture the information about the bugginess of products, and have better performance. As seen in Table 3.10, for GPL, by using the *product implementation* attributes, Clap obtains higher *false-passing* product detection *Recall* than the others. Specifically, only using these attributes, Clap obtained 69.69% in *false-passing* product detection *Recall*, while these figures of Clap with the *test adequacy* attributes and *test effectiveness* attributes are just 53.69% and 50.18%, respectively. For ZipMe, by using the *product implementation* attributes, Clap can correctly detect more than 8 *false-passing* products, while this figure when using the *test adequacy* attributes is only 7. Although, using *test effectiveness* attributes, Clap's *false-passing* product detection *Recall* is better than using the *product implementation*

attributes, CLAP's *true-passing* product detection Recall is much lower, only 48.92%.

Based on only test quality attributes (the *test adequacy* or *test effectiveness*), the model faces difficulties on distinguishing *true-passing* and *false-passing* products. As a result, the CLAP's variants with these attribute have a very high *true-passing* product detection *Recall* but low *false-passing* products detection *Recall* or vice versa. Indeed, test quality is an important factor for CLAP to detect *true-passing* and *false-passing* products. However, the low quality of the test suite is just a sign showing that the test result is less reliable but it cannot confirm the bugginess of the product. Thus, in some cases, using only test quality attributes cannot help to detect *false-passing* products.

As expected, CLAP *obtained the best results when the failure indications are measured based on all three aspects: product implementation, test adequacy, and test effectiveness.* By using all of these attributes, CLAP has more comprehensive information to evaluate the bugginess in the product's source code, as well as the reliability of the product's overall test result. Thus, *all of these attributes should be used together in* CLAP *to achieve the best performance.*

3.6.5 Time Complexity (RQ5)

On average, CLAP took 53 seconds to measure attributes of the passing products of a buggy version (about 2.5 seconds/product). Specifically, each buggy version of ExamDB took only 3 seconds, meanwhile this figure for each version of ZipMe is 192 seconds. Indeed, running time of CLAP depends on the number of sampled products of each system, the number of test cases of each product, and the system's size. The reason is that the proposed attributes are calculated on each passing product of the system. Also, they are measured based on the failure indications investigated from all the failed tests of the failing products of the buggy version. Thus, ExamDB, which contains the least number of sampled products, has the smallest running time. Meanwhile, ZipMe has the largest running time since it is sampled into a large number of products, each product is tested by a large number of test cases, and this is also the largest system.

3.6.6 Threats to Validity

The main threats to the validity of this work are consisted of internal, external, and construct validity threats.

Threats to internal validity mainly lie in the correctness of the ground truth which is labeled for the passing products. To mitigate this threat, this work applied a systematic process to label these products. In the step which requires manual investigation and test generation, I made every effort to carefully investigate the products. Another threat is that the ratio of *false-passing* and *true-passing* in the benchmark might be different from the ratio in practice. To address this threat, I plan to collect real-world *false-passing* and *true-passing* products in larger SPL systems to evaluate the proposed technique.

Threats to external validity mainly lie in the benchmark used in the experiments. Although the dataset uses the systems which are widely used in the existing work, this dataset only contains artificial bugs of Java SPL systems, so the similar results cannot be concluded for real-world faults and SPL systems in other programming languages. In addition, another threat may come from the quality of the products' test suites. For instance, the product could contain low quality test cases (e.g., flaky tests). To mitigate this threat, this work chose the dataset containing a large number of buggy products with a diversity of artificial faults and each products are tested by a large number of test cases. Also, the dataset contains both single-bug and multiple-bug buggy systems. To address these threats, I plan to collect more real-world variability bugs in larger SPL systems to evaluate the proposed technique.

Threats to construct validity mainly lie in the rationality of the assessment metrics. To reduce this threat, this work chose the metrics which are widely used in the related studies [139]. For evaluating the approach of detecting *false-passing* products, this work uses common metrics in classification problem: Precision, Recall, F1-Score, and Accuracy. For evaluating how CLAP helps to mitigate the negative impact of *false-passing* products on FL performance, the most popular metrics in FL studies, Rank and EXAM are used. For assessing the performance of the classifiers, this work follows the instructions from the related studies [139, 144] for choosing the value of $k = 5$ for k-fold cross-validation. Another threat may come from the selected SBFL metrics. To reduce this threat, the five most popular SBFL metrics, which are widely used in FL studies [4, 25], are chosen.

3.7 Summary

In an SPL system, variability bugs can cause failures in certain products (buggy products). However, these buggy products could be incorrectly considered as passing products due to the ineffectiveness of their test suites in detecting the bugs. The buggy products

which still passed all the tests in their test suits are called *false-passing* products. The empirical study has shown that *false-passing* products can produce negative impacts on FL performance. This chapter introduces CLAP, a novel approach to detect *false-passing* products in SPL systems. The key idea of CLAP is that the stronger failure indications in the passing product, the more likely the product is *false-passing*. The failure indications are derived from the failing products of the system based on products' implementation and products' test quality. Then a passing product is measured according to these indications to determine its possibility to be *false-passing*. The experimental results on a large dataset of 823 buggy SPL systems with 14,191 *false-passing* products and 22,555 *true-passing* products show that CLAP can effectively classify *false-passing* and *true-passing* products of the SPL systems, with the average Accuracy of more than 90%. In different experimental scenarios, the precision of *false-passing* product detection by CLAP is up to 96%. Furthermore, this chapter proposes simple and effective methods to mitigate the negative impact of *false-passing* products on fault localization performance. These methods can effectively help the state-of-the-art FL techniques improve their performance by up to 34%. This shows that CLAP can greatly mitigate the negative impact of *false-passing* products on localizing variability bugs and help developers find bugs much faster.

This work was published in the journal of Information and Software Technology in 2023. Nguyen, Thu-Trang, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. "Detecting false-passing products and mitigating their impact on variability fault localization in software product lines." Information and Software Technology 153 (2023): 107080. (ISI/Q1)

Chapter 4

Variability Fault Localization

This chapter presents VarCop, a novel and effective variability FL approach. Firstly, this chapter shows several observations about variability faults. Next, this chapter formulates *feature interaction* which is the root cause of variability failures and introduces an approach to identify suspicious statements. Then, this chapter discusses a solution for measuring the suspiciousness scores of the identified suspicious statements to find the statements that are the most likely to be buggy.

4.1 Introduction

The variability that is inherent to SPL systems challenges quality assurance activities [3, 12–15]. In comparison with the single-system engineering, fault detection and localization through testing in SPL systems are more problematic, as a bug can be *variable*, which can only be exposed under *some* combinations of the system features [12, 16]. Specially, there exists a set of the features that must be selected to be on and off together to necessarily reveal the bug. Due to the presence/absence of the *interaction* among the features in such set, the buggy statements behave differently in the products where these features are on and off together or not. Hence, *the incorrect statements can only expose their bugginess in some particular products, yet cannot in the others.* Specially, in an SPL system, variability bugs only cause failures in certain products, and the others still pass all their tests. This variability property causes considerable difficulties for localizing this kind of bugs in SPL systems.

Despite the importance of *variability fault localization*, the existing FL approaches [4, 6, 25] are not designed for this kind of bugs. These techniques are specialized for finding bugs in a particular product. For instance, to isolate the bugs causing failures in multiple products of a single SPL system, the slice-based methods [25–27] could be used to identify all the failure-related slices for each product independently of others. Consequently, there are multiple sets of (large numbers of) isolated statements that need to be examined to find the bugs. This makes the slice-based methods [25] become impractical in SPL systems.

In addition, the state-of-the-art technique, SBFL [4, 28–31] can be used to calculate the suspiciousness scores of code statements based on the test information (i.e., program spectra) of each product of the system separately. For each product, it produces a ranked list of suspicious statements. As a result, there might be multiple ranked lists produced for a single SPL system which is failed by variability bugs. From these multiple lists, developers cannot determine a starting point to diagnose the root causes of the failures. Hence, it is inefficient to find variability bugs by using SBFL to rank suspicious statements in multiple variants separately.

Another method to apply SBFL for localizing variability bugs in an SPL system is that one can treat the whole system as a single program [5]. This means that the mechanism controlling the presence/absence of the features in the system (e.g., the preprocessor directives `#ifdef`) would be considered as the corresponding conditional `if-then` statements during the localization process. By this adaptation, a single ranked list of the statements for variability bugs can be produced according to the suspiciousness of each statement. Note that, this work considers the product-based testing [32, 33]. Specially, each product is considered to be tested individually with its own test set. Additionally, a test, which is designed to test a feature in domain engineering, is concretized to multiple test cases according to products' requirements in application engineering [32]. Using this adaptation, the suspiciousness of the statement is measured based on the total numbers of the passed and failed tests executed by it in all the tested products. Meanwhile, the characteristics including the interactions between system features and the variability of failures among products are also useful to isolate and localize variability bugs in SPL systems. However, these kinds of important information are not utilized in the existing approaches.

This chapter proposes VarCop, a novel fault localization approach for variability bugs. The key ideas of VarCop is that variability bugs are localized based on (i) the interaction among the features which are necessary to reveal the bugs, and (ii) the bugginess exposure which is reflected via both the overall test results of products and the detailed result of each test case in the products.

For a buggy SPL system, VarCop detects sets of the features which need to be selected on/off together to make the system fail by analyzing the overall test results (i.e., the state of passing all tests or failing at least one test) of the products. This dissertation calls each of these sets of the feature selections a *Buggy **P**artial **C**onfiguration (Buggy PC)*. Then, VarCop analyzes the interaction among the features in these *Buggy PCs* to isolate

the statements which are suspicious. In VarCop, the suspiciousness of each isolated statement is assessed based on two criteria. The first criterion is based on the overall test results of the products containing the statement. By this criterion, the more failing products and the fewer passing products where the statement appears, the more suspicious the statement is. Meanwhile, the second one is assessed based on the suspiciousness of the statement in the failing products which contain it. Specially, in each failing product, the statement's suspiciousness is measured based on the detailed results of the products' test cases. The idea is that if the statement is more suspicious in the failing products based on their detailed test results, the statement is also more likely to be buggy in the whole system.

This work conducted experiments to evaluate VarCop in both single-bug and multiple-bug settings on a dataset of 1,570 versions (cases) containing variability bug(s) [5]. The performance of VarCop is compared with the state-of-the-art approaches including (SBFL) [4, 28–31], the combination of the slicing method and SBFL (S-SBFL) [120, 121], and Arrieta et al. [6] using 30 most popular SBFL ranking metrics [4, 28, 29]. The experimental results show that VarCop significantly outperformed the baselines in all the studied metrics.

4.2 Motivating Example

This section illustrates the challenges of localizing variability bugs and motivate the solution via an example.

4.2.1 An Example of Variability Faults in Software Product Lines

Listing 4.1 shows a simplified variability bug in *Elevator System* [5]. The overall test results of the sampled products are shown in Table 4.1. In Listing 4.1, the bug (incorrect statement) at line 31 causes the failures in products p_6 and p_7.

Listing 4.1: An example of variability fault in Elevator System

```
1  int maxWeight = 2000, weight = 0;
2
3  //#ifdef Empty
4  void empty(){ persons.clear();}
5  //#endif
6  void enter(Person p){
```

Table 4.1: The sampled products and their overall test results

P	C	*Base*	*Empty*	*Weight*	*TwoThirdsFull*	*Overloaded*
p_1	c_1	T	F	T	F	F
p_2	c_2	T	T	T	F	F
p_3	c_3	T	T	F	F	F
p_4	c_4	T	F	T	T	F
p_5	c_5	T	F	T	T	T
p_6	c_6	T	T	T	F	T
p_7	c_7	T	F	T	F	T

P and C are the sampled sets of products and configurations.
p_6 and p_7 fail at least one test (*failing products*). Other products pass all their tests (*passing products*).

```
7       persons.add(p);
8       //#ifdef Weight
9       weight += p.getWeight();
10      //#endif
11  }
12  void leave(Person p){
13      persons.remove(p);
14      //#ifdef Weight
15      weight -= p.getWeight();
16      //#endif
17  }
18  ElevState stopAtAFloor(int floorID){
19      ElevState state = Elev.openDoors;
20      boolean block = false;
21      for (Person p: new ArrayList<Person>(persons))
22          if (p.getDestination() == floorID)
23              leave(p);
24      for (Person p : waiting) enter(p);
25      //#ifdef TwoThirdsFull
26      if (weight >= maxWeight*2/3)
27          block = true;
28      //#endif
29      //#ifdef Overloaded
30      if(block == false){
31          if (weight == maxWeight )
32              //Patch: weight >= maxWeight
33              block = true;
34      }
35      //#endif
36      if (block == true)
37          return Elev.blockDoors;
38      return Elev.closeDoors;
39  }
```

In this system, the total loaded weight of the Elevator cabin is guaranteed under a safety bound by *TwoThirdsFull* and *Overloaded*. *TwoThirdsFull* is expected to limit the load not to exceed 2/3 of the elevator's capacity, while *Overloaded* ensures the maximum load is the elevator's capacity. However, the implementation of *Overloaded* (lines 30–34) does not

behave as specified. If the total loaded weight (`weight`) of the elevator is tracked, then instead of blocking the elevator when `weight` exceeds its capacity (`weight >= maxWeight`), its actual implementation blocks the elevator **only** when `weight` is equal to `maxWeight` (line 31). Consequently, if *Weight* and *Overloaded* are on (and *TwoThirdsFull* is off), even the total loaded weight is greater than the elevator's capacity, then (`block==false`) the elevator still dangerously works without blocking the doors (lines 37–39).

This bug (line 31) is *variable* (*variability bug*). It is revealed not in all the sampled products, but only in p_6 and p_7 (Table 4.1) due to the *interaction* among *Weight*, *Overloaded*, and *TwoThirdsFull*. Specially, the behavior of *Overloaded* which sets the value of `block` at line 33 is interfered by *TwoThirdsFull* when both of them are on (lines 27 and 30). Moreover, the incorrect condition at line 31 can be exposed only when *Weight* = *T*, *TwoThirdsFull*=*F*, and *Overloaded* = *T* in p_6 and p_7. Hence, understanding the root cause of the failures to localize the variability bug could be very challenging.

4.2.2 Observations

For an SPL system containing variability bugs, there are certain features that are (ir)relevant to the failures [12, 76, 132]. In Listing 4.1, enabling or disabling feature *Empty* does not affect the failures. Indeed, some products still fail (p_6 and p_7) or pass their test cases (p_1 and p_2) regardless of whether *Empty* is on or off. Meanwhile, there are several features which **must be enabled/disabled together** to make the bugs visible. In other words, for certain products, changing their current configurations by switching the current selection of anyone in these relevant features makes the resulting products' overall test results change. For *TwoThirdsFull*, switching its current off-selection in the failing product, p_7, makes the resulting product, p_5, where $TwoThirdsFull = T$, behave as expected (Table 4.1). The reason is that in p_5, the presence of *Weight* and *TwoThirdsFull* impacts *Overloaded*, and consequently *Overloaded* does not expose its incorrectness. In fact, this characteristic of variability bugs has been confirmed by the public datasets of real-world variability bugs [76, 132]. For example, in the VBDb [76], there are 41/98 bugs revealed by a single feature and the remaining 57/98 bugs involved 2–5 features. The occurrence condition of these bugs is relevant to both enabled and disabled features. Particularly, 49 bugs occurred when all of the relevant features must be enabled. Meanwhile, the other half of the bugs require that at least one relevant feature is disabled.

The impact of features on each other is their **feature interaction** [12, 69]. The presence

of a feature interaction makes certain statements behave differently from when the interaction is absent. For variability bugs, the presence of the interaction among the relevant features[1] exposes/conceals the bugginess of the statements that cause the unexpected/-expected behaviors of the failing/passing products [12, 76]. Thus, to localize variability bugs, it is necessary to identify such sets of the relevant features as well as the interaction among the features in each set, which is the root cause of the failures in failing products.

In an SPL system containing variability bugs, there might be multiple such sets of the relevant features. Let consider a particular set of relevant features.

O1. *In the features F_E which **must be enabled together** to make the bugs visible, the statements that implement the interaction among these features provide valuable suggestions to localize the bugs.* For instance, in Listing 4.1, F_E consists of *Weight* and *Overloaded*, and the interaction between these features contains the buggy statement at line 31 (s_{31}) in *Overloaded*. This statement uses variable `weight` defined/updated by feature *Weight* (lines 9 and 15). Hence, detecting the statements that implement the interaction among the features in F_E could provide us valuable indications to localize the variability bugs in the systems.

O2. Moreover, *in the features F_D which **must be disabled together** to reveal the bugs, the statements impacting the interaction among the features in F_E (if the features in F_D and F_E are all on), also provide useful indications to help us find bugs.* In Listing 4.1, although the statements from lines 26–27 in *TwoThirdsFull* (being disabled) are not buggy, analyzing the impact of these statements on the interaction between *Overloaded* and *Weight* can provide the suggestion to identify the buggy statement. The intuition is that the features in F_D have the impacts of "hiding" the bugs when these features are enabled. In this example, when *Weight*, *TwoThirdsFull* and *Overloaded* are all on, if the loaded weight exceeds `maxWeight*2/3` (i.e., the conditions at line 26 are satisfied), then `block = true`, and the statements from 31–33 (in *Overloaded*) cannot be executed. As a result, the impact of the incorrect condition at line 31 is "hidden". Thus, the impact of the features in F_D (as if they are on) on the interaction among the features in F_E should be considered in localizing variability bugs.

O3. For a buggy SPL system, a statement can appear in both failing and passing products. Meanwhile, the states of failing or passing of the products expose the bugginess of the

[1] Without loss of generality, for the cases where there is only one relevant feature, the feature can impact and interact itself.

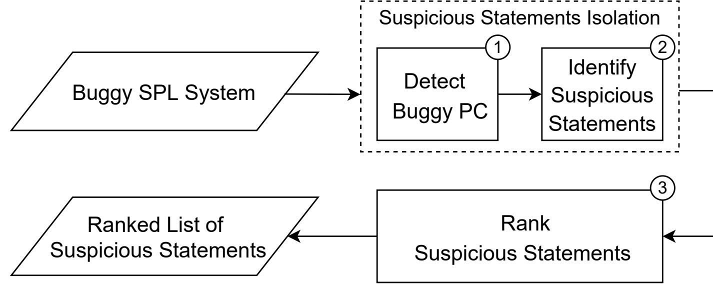


Figure 4.1: VarCop's Overview

contained buggy statements. Thus, *the overall test results of the sampled products can be used to measure the suspiciousness of the statements.* Furthermore, the bugginess of an incorrect statement can be exposed via the detailed (case-by-case) test results in every failing product containing the bug. In this example, s_{31} is contained in two failing products p_6 and p_7, and its bugginess is expressed differently via the detailed test results in p_6 and p_7. Thus, to holistically assess the suspiciousness of a statement s, *the score of s should also reflect the statement's suspiciousness via the detailed test results in every failing product where s contributes to.*

Among these observations, **O1** and **O2** will be theoretically discussed in Section 4.3.2. Also, the observation **O3** is empirically validated by the experiments (Section 4.8).

4.2.3 VarCop Overview

In this dissertation, the problem of variability fault localization is defined in Definition 4.1.

Definition 4.1 ***(Variability Fault Localization).*** *Given 3-tuple $\langle \mathfrak{S}, P, R \rangle$, where:*

- $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$ *is a system containing variability faults,*
- $P = \{p_1, p_2, ..., p_n\}$ *is the set of sampled products,* $P = P_P \cup P_F$*, where* P_P *and* P_F *are the sets of passing and failing products of* $\mathfrak{S}$*, and*
- $R = \{R_1, R_2, ..., R_n\}$ *is the set of the program spectra of the products in* P*, where* R_i *is the program spectrum of* p_i*.*

Variability Fault Localization is to output the list of the statements in $\mathfrak{S}$ ranked based on their suspiciousness to be buggy.

Based on the observations shown in Section 4.2.2, this chapter proposes VarCop, a novel variability fault localization approach. For a given buggy SPL system, the input of VarCop consists of a set of the tested products and their program spectra. VarCop outputs a ranked list of suspicious statements in three steps (Figure 4.1):

1. First, by analyzing the configurations and the overall test results of the sampled products, VarCop detects minimal sets of features whose *selections* (the states of being *on/off*) make the bugs (in)visible. Let us call such a set of selections a *Buggy Partial Configuration (Buggy PC)*. In Listing 4.1, $\{Weight = T$, $Overloaded = T$, $TwoThirdsFull = F\}$ is a *Buggy PC*.

2. Next, for each failing product, VarCop isolates the *suspicious statements* which are responsible for implementing the interaction among the features in each detected *Buggy PC*. Specially, the feature interaction implementation is a set of the statements which these features use to *impact* each other. For example, in p_7, VarCop analyzes its code to detect the implementation of the interaction among *Weight*, *Overloaded*, and *TwoThirdsFull* (**O1** and **O2**), and this interaction implementation includes the statements at lines 9, 15, and 31. Intuitively, all the statements in p_7 which have an impact on these statements or are impacted by them are also suspicious.

3. Finally, the suspicious statements are ranked by examining how their suspiciousness exposes in both the overall test results of the containing products (*product-based suspiciousness assessment*) and these products' detailed case-by-case test results (*test case-based suspiciousness assessment*). Particularly, for each isolated statement, the product-based assessment is calculated based on the numbers of the passing and failing products containing the statement. Meanwhile, the test case-based suspiciousness is assessed by aggregating the suspiciousness scores of the statement in the failing products which are calculated based on the detailed results of the tests executed by the statement. (**O3**).

4.3 Feature Interaction

This section introduces the feature interaction formulation which can be used to analyze the root cause of variability faults.

4.3.1 Feature Interaction Formulation

Different kinds of feature interactions have been discussed in the literature [12, 15, 69, 74]. This work formulates feature interaction based on the impacts of a feature on other features. Specially, for a set of features in a product, a feature can interact with the others in two ways: (i) directly impacting the others' implementation and (ii) indirectly impacting the others' behaviors via the statements which are impacted by *all* of them. For (i), there is control/data dependency between the implementation of these features. For example, in p_5, since the statement at line 26 (s_{26}) in $\varphi(TwoThirdsFull)$ is data-dependent on s_9 and s_{15} in $\varphi(Weight)$, there is an interaction between $Weight$ and $TwoThirdsFull$ in p_5. For (ii), there is at least one statement which is control/data dependent on a statement(s) of every feature in the set. For instance, in p_5, $TwoThirdsFull$ and $Weight$ interact by both impacting the statement at line 31. As a result, when these features are all on in a product, each of them will impact the others' behaviors.

Without loss of generality, a statement can be considered to be impacted by that statement itself. Thus, for a set of enabled features F, in a product, there exists an interaction among these features if there is a statement s which is impacted by the implementation of all the features in F, regardless of whether s is used to implement these features or not. Formally, this work defines *impact function*, Ω, to determine the impact of a statement in a product.

Definition 4.2 ***(Impact Function).*** *Given a system* $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$*, the impact function is defined as* $\Omega : \mathbb{S} \times \mathbb{P} \to 2^{\mathbb{S}}$*. Specially,* $\Omega(s, p)$ *refers to the set of the statements of* $\mathfrak{S}$ *which are impacted by statement* s *in product* p*. For a statement* s' *in product* p*,* $s' \in \Omega(s, p)$ *if* s' *satisfies one of the following conditions:*

- $s' = s$
- s' *is data/control-dependent on* s

For the example, $\Omega(s_{33}, p_5) = \{s_{33}, s_{36}, s_{37}, s_{38}\}$. Note that if statement s is not in product p, then $\Omega(s, p) = \emptyset$.

Definition 4.3 ***(Feature Interaction).*** *Given a system* $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$*, for product* p *and a set of features* F *which are enabled in* p*, the interaction among the features in* F

exists if and only if the following condition is satisfied:

$$\bigcap_{f \in F} \alpha(f, p) \neq \emptyset$$

where $\alpha(f, p) = \bigcup_{s \in \varphi(f)} \Omega(s, p)$ *refers to the set of the statements in* p *which are impacted by any statement in* $\varphi(f)$*. The* ***implementation of the interaction*** *among features* F *in product* p *is denoted by* $\beta(F, p) = \bigcap_{f \in F} \alpha(f, p)$*.*

For the example in Listing 4.1, the features $Weight$ and *Overloaded* interact with each other in product p_7 and the implementation of the interaction includes the statements at lines 31–33 and 36–38. Note that, without loss of generality, a feature can impact and interact with itself.

4.3.2 The Root Cause of Variability Failures

This section analyzes and discusses the relation between variability failures in SPL systems and the enabling/disabling of features. In a buggy SPL system, the variability bugs can be revealed by set(s) of the **relevant** features which must be enabled and disabled **together** to make the bugs visible. For each set of relevant features, their selections might affect the visibility of the bugs in the system. For simplicity, this section first analyzes the buggy system containing a single set of such relevant features. The cases where multiple sets of relevant features involve in the variability failures will be discussed in the later part.

Let consider the cases where the failures of a system are revealed by *a single set of the relevant features*, $F_r = F_E \cup F_D$, where the features in F_E and F_D must be respectively enabled and disabled **together** to make the bugs visible. Specially, the features in F_E and F_D must be respectively on and off in all the failing products. From a failing product $p \in P_F$, once switching the current selection of any switchable feature[2] in F_r, the resulting product p' will pass all its tests, $p' \in P_P$. In this case, the interaction of features in F_r propagates the impact of buggy statements to the actual outputs causing the failures in p. The relation between variability bugs *(buggy statements)* causing the failures in failing products and the interaction between the relevant features in F_r will be theoretically discussed as following.

[2]In configuration c, feature f is switchable if switching f's selection, the obtained configuration is valid regarding system's feature model.

For a failing product $p \in P_F$, the set of the buggy statements in p is denoted by $\mathcal{S}_b$. From p, disabling any feature $f_e \in F_E$ would produce a passing product $p' \in P_P$. In this case, **every** buggy statement $s \in \mathcal{S}_b$ can be either present or not in p'. First, if s is not in p' after disabling f_e from p, then $s \in \varphi(f_e) \subseteq \alpha(f_e, p)$. The second case is that s is still in p'. Due to the absence of f_e, s *behaves differently* from the way it does incorrectly in p, and p' passes all its tests. This means, in p, f_e has impact on s and/or the statements impacted by s, $\Omega(s, p)$. In other words, $\alpha(f_e, p) \cap \Omega(s, p) \neq \emptyset$. In p, s and the statements impacted by s together propagate their impacts to the unexpected outputs in p. Thus, any change on the statements in $\Omega(s, p)$ can affect the bugs' visibility.

These two above cases show that *every* incorrect statement s in $\mathcal{S}_b$, only exposes its bugginess with the presence of *all* the features in F_E. This demonstrates that the features in F_E must interact with each other in p, $\beta(F_E, p) = \bigcap_{f_e \in F_E} \alpha(f_e, p) \neq \emptyset$. Indeed, if there exists a feature $f \in F_E$ which does not interact with the others in F_E, $\alpha(f, p) \cap \beta(F_E \setminus \{f\}, p) = \emptyset$, the incorrect behaviors of s will only be impacted by either f or the interaction among the features in $\{F_E \setminus f\}$. As a result, s will not require the presence of both f and $\{F_E \setminus f\}$ to reveal its bugginess. Moreover, since every $f_e \in F_E$ has impacts on the behaviors of all the buggy statements, $\forall s \in \mathcal{S}_b$, $\forall f_e \in F_E, \Omega(s, p) \cap \alpha(f_e, p) \neq \emptyset$, the interaction among the features in F_E also has impacts on the behaviors of every buggy statement, $\forall s \in \mathcal{S}_b$, $\Omega(s, p) \cap \beta(F_E, p) \neq \emptyset$. In other words, the features F_E interact with each other in p, and the interaction implementation impacts the visibility of the failures caused by every buggy statement. Hence, the statements which implement the interaction among the features in F_E are valuable suggestions to localize the buggy statements. This theoretically confirms the observation **O1**.

Similarly, from p, turning on any disabled feature $f_d \in F_D$, the resulting product p'' also passes all its tests. This illustrates that in p'', the behaviors of **every** buggy statement s in $\mathcal{S}_b$ are impacted by the presence of f_d, thus the incorrect behaviors of s cannot be exposed. Formally, $\forall s \in \mathcal{S}_b, \Omega(s, p) \cap \alpha(f_d, p'') \neq \emptyset$. Intuitively, f_d has impacts on interaction implementation of F_E in p as well as the impact of this interaction on $\Omega(s, p)$. In other words, *the interaction among $F_E \cup F_D$ impacts the behaviors of the buggy statements,* $\forall s \in \mathcal{S}_b$*,* $\Omega(s, p) \cap \beta(F_E \cup F_D, p'') \neq \emptyset$. Hence, investigating the interaction implementation of the features in F_D and F_E (if they were all enabled in a product) can provide us useful indications to find the incorrect statements. This explains the observation **O2**.

Overall, the interaction among the relevant features in $F_r = F_E \cup F_D$ reveals/hides the bugs by impacting the buggy statements and/or the statements impacted by the buggy ones in the products. Illustratively, this interaction implementation propagates the impact of all the buggy statements to the output of the failed tests in a failing product. This means that the buggy statements are contained in the set of statements which are impacted by or have impacts on the interaction implementation of the relevant features. Thus, *identifying the sets of the relevant features whose interaction can affect the variability bugs visible/invisible and the implementation of the interaction is necessary to localize variability bugs in SPL systems.*

In general, variability bugs in a system can be revealed by multiple sets of relevant features. In these cases, the visibility of the bugs might not be clearly observed by switching the selections of features in one set of relevant features. For instance, F_r and F'_r are two sets of relevant features whose interaction causes the failures in the system. Once switching the selection of any feature in F_r, the implementation of the interaction among the features in F_r is not in the resulting product p'. Meanwhile, p' can still contain the interaction among the features in F'_r. Thus, p' can still fail some tests. However, if F_r and/or F'_r or even their subsets can be identified, by examining the interaction among the identified set(s) of features, the bugs can be isolated. More details will be described and proved in the next section.

In spite of the importance of the relation between variability failures and relevant features, this information is not utilized by existing studies such as SBFL and S-SBFL. Consequently, their resulting suspicious spaces are often large. Meanwhile, by Arrieta et al. [6], SBFL is adapted to localize bugs at the feature-level. Particularly, each sampled product is considered as a test (i.e., passed tests are passing products, and failed tests are failing products), and the spectra record the feature selections in each product. However, SBFL is used to localize the *buggy features*. By this method, all the statements in the same feature have the same suspiciousness level. Thus, this approach could be ineffective for localizing variability bugs at the statement-level. This will be empirically illustrated in the experimental results (Section 4.8).

4.4 Buggy Partial Configuration Detection

This section introduces the notions of *Buggy Partial Configuration (Buggy PC)* and *Suspicious Partial Configuration (Suspicious PC)*. Specially, *Buggy PCs* are the partial config-

urations whose interactions among the corresponding features are the root causes making variability bugs visible in a buggy system. In general, *Buggy PCs* can be detected after testing all the possible products of the system. However, verifying all those products is infeasible in practice. Meanwhile, *Suspicious PCs* are the detected suspicious candidates for the *Buggy PCs* which can be practically computed using the sampled products.

4.4.1 Buggy Partial Configuration

For a buggy system $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$, where all the possible configurations of $\mathfrak{S}$, $\mathbb{C}$, is categorized into the non-empty sets of passing ($\mathbb{C}_P$) and failing ($\mathbb{C}_F$) configurations, $\mathbb{C}_P \cup \mathbb{C}_F = \mathbb{C}$, a Buggy **P**artial **C**onfiguration (*Buggy PC*) is the minimal set of feature selections that makes the bugs visible in the products. In Listing 4.1, the only *Buggy PC* is $B = \{Weight = T, TwoThirdsFull = F, Overloaded = T\}$.

Definition 4.4 ***(Buggy Partial Configuration (Buggy PC)).*** *Given a buggy system $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$, a buggy partial configuration, B, is a set of feature selections in $\mathfrak{S}$ that has both the following Bug-Revelation and Minimality properties:*

- ***Bug-Revelation**. Any configuration containing B is corresponding to a failing product, $\forall c \in \mathbb{C}, c \supseteq B \implies c \in \mathbb{C}_F$.*
- ***Minimality**. There are no strict subsets of B satisfying the Bug-Revelation property, $\forall B' \subsetneq B \implies \neg(\forall c \in \mathbb{C}, c \supseteq B' \implies c \in \mathbb{C}_F)$.*

Bug-Revelation. This property is equivalent to that *all the passing configurations do not contain B*. Indeed,

$$\forall c \in \mathbb{C}, c \supseteq B \implies c \in \mathbb{C}_F$$

$$\Leftrightarrow \forall c \in \mathbb{C}, [c \not\supseteq B \vee c \in \mathbb{C}_F]$$

$$\Leftrightarrow \forall c \in \mathbb{C}, [c \not\supseteq B \vee c \notin \mathbb{C}_P]$$

$$\Leftrightarrow \forall c \in \mathbb{C}, c \in \mathbb{C}_P \implies c \not\supseteq B$$

For a set of feature selections B', if there exists a passing configuration containing B', $\exists c \in \mathbb{C}_P \wedge c \supset B'$, the interaction among the features in B' in the corresponding product

p cannot be the root cause of any variability bug. This is because there is no unexpected behavior caused by this interaction in the passing product p[3]. Hence, investigating the interaction between them *might not* help us localize the bugs. For example, $B' = \{Empty = T, Weight = T\}$ is a subset of the failing configuration c_6, however it is not considered as a *Buggy PC*, because B' also is a subset of c_2 which is a passing configuration. Indeed, in every product, the interaction between *Empty* and $Weight$, which does not cause any failure, should not be investigated to find the bug. Thus, to guarantee that the interaction among the features in a *Buggy PC* is the root cause of variability bugs, the set of feature selections needs to have *Bug-Revelation* property.

Minimality. If a set B holds the *Bug-Revelation* property but not *minimal*, then there exists a strict subset B' of B ($B' \subsetneq B$) that also has *Bug-Revelation* property. However, for any $p \in P_F$ whose configuration contains both B and B', to detect all the bugs related to either B or B', the smaller one, B', should still be examined rather than B. The reason is, in p, the bugs related to both B and B' are all covered by the implementation of the interaction among the features in B'.

Particularly, let $B' = F'_E \cup F'_D$ and $B = F_E \cup F_D$, where F'_E, F'_D, F_E, and F_D are the sets of enabled and disabled features in B' and B, respectively. Since $B' \subset B$, and $F'_E \cap F'_D = F_E \cap F_D = \emptyset$, so $F'_E \subseteq F_E$ and $F'_D \subseteq F_D$. For the enabled features in F_E and F'_E, the failures in p can be caused by the interactions among the enabled features in F_E or F'_E. The implementation of the interactions among the enabled features in F_E and F'_E in p are $\beta(F_E, p) = \bigcap_{f \in F_E} \alpha(f, p)$ and $\beta(F'_E, p) = \bigcap_{f' \in F'_E} \alpha(f', p)$, respectively. Then, $\beta(F_E, p) \subseteq \beta(F'_E, p)$ because $F'_E \subseteq F_E$. As a result, the incorrect statements related to $\beta(F_E, p)$ are all included in $\beta(F'_E, p)$. Similarly, for the sets of the disabled features, the set of the statements in p related to all the features of F'_D also includes the statements related to all the features of F_D. In consequence, by identifying the interaction implementation of the features in B', the bugs which are related to both B and B' can all be found.

Furthermore, if both B and B' are related to the same bug(s), the larger set could contain bug-irrelevant feature selections which can negatively affect the FL effectiveness. For example, both the entire configuration c_6 and its subset $\{TwoThirdsFull = F, Overloaded = T, Weight = T\}$ has *Bug-Revelation* property. Nevertheless, the interaction among $TwoThirdsFull$, *Overloaded*, and $Weight$, which indeed causes the failures

[3]Assuming that the test suite of each product is effective in detecting bugs. This means the buggy products must fail.

in p_6, should be investigated instead of the interaction among the features in the entire c_6. This configuration contains the bug-irrelevant selection, *Empty* $= T$. As a result, *Buggy PCs* need to be *minimal*.

***Buggy PC* Detection Requirement**. All possible configurations of a system are very rarely available for the QA process. Thus, in a buggy system, the sampled set is used to detect the *Buggy PCs*. Assuming that $\mathbb{D}$ is the set of the candidates for *Buggy PCs* which is detected by an FL approach. For any $D \in \mathbb{D}$, all the statements which implement the interaction among the features in D and the statements which have impact on that interaction implementation (Section 4.3.2) are suspicious. Thus, for a *Buggy PC* B, to avoid missing related buggy statements $\mathcal{S}$, the FL method must ensure that $\mathcal{S}$ is covered by the suspicious statements identified from at least one candidate: *there must be at least one candidate in* $\mathbb{D}$ *which is a subset of* B, $\exists D \in \mathbb{D}$, $D \subseteq B$. Let us call this the **effectiveness requirement**.

Indeed, if there exists $D \in \mathbb{D}$, such that $D \subseteq B$, then in a product p whose configuration contains B (apparently contains D), the interaction implementation of the features in D covers all the statements implementing the interaction of the features in B, $\beta(D,p) \supseteq \beta(B,p)$. As a result, the suspicious statements set of D contains both the interaction implementation $\beta(B,p)$ and the statements which have impact on this interaction in p. In other words, the suspicious statements set of D contains buggy statements $\mathcal{S}$. Hence, to guarantee the effectiveness in localizing variability bugs, this work aims to detect the set of the candidates for *Buggy PCs* which satisfies the effectiveness requirement.

4.4.2 Important Properties to Detect Buggy Partial Configuration

In practice, a system $\mathfrak{S}$ may have a huge number of possible configurations, $\mathbb{C}$. Consequently, only a subset of $\mathbb{C}$ is sampled for testing and debugging, $C = C_P \cup C_F$. A set of feature selections which has both *Bug-Revelation* and *Minimality* properties on the sampled set C is intuitively *suspicious* to be a *Buggy PC*. Let us call these sets of selections *Suspicious Partial Configurations (Suspicious PCs)*.

For example, in Listing 4.1, $D = \{TwoThirdsFull = F, Overloaded = T\}$ is a *Suspicious PC*. All the configurations containing D (c_6 and c_7) are failing. Additionally, all the strict subsets of D do not hold *Bug-Revelation* on C, e.g., $\{TwoThirdsFull = F\}$ is in c_1, and $\{Overloaded = T\}$ is in c_5, which are passing configurations. Thus, D is a minimal set which holds the *Bug-Revelation* property on C.

Theoretically, the *Suspicious PCs* in c can be detected by examining all of its subsets to identify the sets satisfying both *Bug-Revelation* and *Minimality.* The number of sets that need to be examined could be $2^{|c|} - 1$. However, not every selection in c participates in *Suspicious PCs*. Hence, to detect *Suspicious PCs* efficiently, this work aims to identify a set of the selections, SFS_c, (*Suspicious Feature Selections*) of c in which the selections potentially participate in one or more *Suspicious PCs*. Then, instead of inefficiently examining all the possible subsets of c, the subsets of SFS_c, which is a subset of c, are inspected to identify *Suspicious PCs*.

Particularly, in failing configuration c, there exist the selections such that switching their current states (from *on* to *off*, or vice versa) results in a passing configuration c'. In other words, the bugs in the product of c are invisible in the product corresponding to c', and the resulting product passes all its tests. Intuitively, each of these selections might be relevant to the visibility of the bugs. Each of these selections can be considered as a *Suspicious Feature Selection.* Thus, a selection, which is in a failing configuration yet not in a passing one, is suspicious to the visibility of the bugs.

Definition 4.5 ***(Suspicious Feature Selection (SFS)).*** *For a failing configuration $c \in C_F$, a feature selection $f_s \in c$ is suspicious if f_s is not present in at least one passing configuration, formally $\exists c' \in C_P, f_s \notin c'$, specially, $f_s \in c \setminus c'$.*

For example, the SFS_7 of c_7 contains the selections in the set differences of c_7 and the passing configurations, e.g., $c_7 \setminus c_1 = \{Overloaded = T\}$. Intuitively, the set difference $(c_7 \setminus c_1)$ must contain a part of every *Buggy PC* in c_7, otherwise c_1 would contain a *Buggy PC* and fail some tests. Hence, any failing configuration c has the following property about the relation between the set of the *Buggy PCs* in c and $(c \setminus c')$ where c' is a passing configuration.

Property 4.1 *Given failing configuration c whose set of the Buggy PCs is BPC_c, the difference of c from any passing configuration c' contains a part of every Buggy PC in BPC_c. Formally, $\forall c' \in C_P, \forall B \in BPC_c, (c \setminus c') \cap B \neq \emptyset$.*

The intuition is that for a failing configuration c and a passing one c', a *Buggy PC* in c can be either in their common part $(c \cap c')$ or their difference part $(c \setminus c')$. As c' is a passing configuration, the difference must contain a part of every *Buggy PC* in c. Otherwise,

there would exist a *Buggy PC* in both c and c', and c' should be a *failing* configuration (*Bug-Revelation* property). This is impossible.

For a configuration $c \in C_F$, SFS_c is denoted as the set of all the suspicious feature selections of c, $SFS_c = \bigcup_{c' \in C_P}(c \setminus c')$. To detect *Buggy PCs* in c, VarCop identifies all subsets of SFS_c which satisfy the *Bug-Revelation* and *Minimality* properties regarding to C. The following property demonstrates that the proposed method maintains the *effectiveness requirement* in detecting *Buggy PCs* (Section 4.4.1).

Property 4.2 *Given a failing configuration c whose set of the Buggy PCs is BPC_c, for any $B \in BPC_c$, there exists a subset of SFS_c, $M \subseteq SFS_c$, such that M satisfies the Bug-Revelation condition in the sampled set C and $M \subseteq B$.*

Proof 4.1 *Considering a Buggy PC of a failing configuration c, $B \in BPC_c$, $(c \setminus c') \cap B$ is denoted by $M_{c'} \neq \emptyset$, where c' is a passing configuration (Property 4.1). Let consider $M = \bigcup_{c' \in C_P} M_{c'}$. Note that, as $M_{c'} \subseteq (c \setminus c')$, thus, $M \subseteq SFS_c = \bigcup_{c' \in C_P}(c \setminus c')$. In addition, since $M_{c'} \subset B$ for every passing configuration $c' \in C_P$, so $M \subset B$. Moreover, for every passing configuration $c' \in C_P$, because $c' \not\supset M_{c'}$ (since $M_{c'} \subset (c \setminus c')$), c' does not contain any superset of M_c, then $c' \not\supset M$. As a result, M, which is a subset of both SFS_c and B, satisfies Bug-Revelation property.*

As a result, given $c \in C_F$, there always exists a common subset of SFS_c and B, for any $B \in BPC_c$, and that set satisfies *Bug-Revelation* property on C. Hence, according to the *effectiveness requirement* (Section 4.4.1), detecting *Buggy PCs* by examining the subsets of SFS_c is effective to localize the variability bugs. Furthermore, as SFS_c only contains the differences of c and other passing configurations, $|SFS_c| \leq |c|$. For example in Table 4.1, *SFSs* of configuration c_6 is $SFS_6 = \{Empty = T, Weight = T, TwoThirdsFull = F, Overloaded = T\}$, so $|SFS_6| < |c_6|$. Thus, SFS_c should be used to detect *Buggy PCs* rather than c.

Note that *Suspicious PCs* and *Buggy PCs* all hold the *Bug-Revelation* on the given sampled set of configurations, C. However, because there are some (passing) configurations in $\mathbb{C}$ but not in C, it does not express that some selections must be in a *Buggy PC*. Hence, *Buggy PCs might not hold Minimality property on C*. In Listing 4.1, a *Buggy PC* is $B = \{Weight = T, TwoThirdsFull = F, Overloaded = T\}$. However, B does not satisfy *Minimality* on the set of the available configurations (Table 4.1), because its

subset, $\{TwoThirdsFull = F, Overloaded = T\}$, satisfies *Bug-Revelation* on C. The reason is that Table 4.1 does not show any product whose configuration contains both $\{Weight = F\}$ as well as $\{TwoThirdsFull = F, Overloaded = T\}$ passes or fails their tests. Therefore, $\{Weight = T\}$ is not expressed as a part of the *Buggy PC*.

4.4.3 Buggy Partial Configuration Detection Algorithm

Algorithm 4.1 shows an algorithm to detect *Buggy PCs* and return the *Suspicious PCs* in a buggy system, given the sets of the passing and failing configurations, C_P and C_F.

In Algorithm 4.1, all the *Suspicious PCs* in the system are collected from the *Suspicious PCs* identified in each failing configuration c (lines 3–19). From lines 5–9, the set of the suspicious selections in c is computed. In order to do that, the differences of c from all the passing configurations are gathered and stored in SFS_c (lines 6–9).

Next, the *Suspicious PCs* in c are the subsets of SFS_c which have both *Bug-Revelation* and *Minimality* with respect to $C = C_F \cup C_P$ (lines 13–15). Each candidate, a set of feature selections *cand*, is checked against these properties by $satisfy$ (line 13). In Algorithm 4.1, the examined subsets of SFS_c have the maximum size of K (lines 10–11). In other words, the considered interactions are up to K-way. In practice, most of the bugs are caused by the interactions of *fewer than 6 features* [12, 77]. Thus, one should set $K = 7$ to ensure the efficiency. Specially, the function $subsetWithSize(SFS_c, k)$ (line 11) returns all the subsets size k of SFS_c. Note that if a set is already a *Suspicious PC*, then any superset of it would not be a *Suspicious PC* (violating *Minimality*). Thus, all the supersets of the identified *Suspicious PCs* can be early eliminated (line 12).

In the example, from Table 4.1, the detected *Suspicious PCs* are two sets $D_1 = \{Empty = T, Overloaded = T\}$ and $D_2 = \{TwoThirdsFull = F, Overloaded = T\}$.

4.5 Suspicious Statement Identification

For a buggy SPL system, the incorrect statements can be found by examining the statements which implement interactions of *Buggy PCs* as well as the statements impacting that implementation, as discussed in Section 4.3.2. Thus, all the statements which implement the interactions of *Suspicious PCs* and the statements impacting them, are considered as suspicious to be buggy. In a product $p \in P_F$, for a *Suspicious PC* D whose the sets of enabled and disabled features are F_E and F_D, respectively. Hence, in p, the

Algorithm 4.1: *Buggy PC* Detection Algorithm

```
1  Procedure DetectBuggyPCs(C_P, C_F)
     Input  : C_P is the set of passing configurations
              C_F is the set of failing configurations
     Output: Set of suspicious partial configurations (SuspiciousPCSet)
2    begin
3      SuspiciousPCSet ← ∅
4      for c ∈ C_F do
5        SFS_c ← ∅
6        for c' ∈ C_P do
7          F_S ← c \ c'
8          SFS_c ← SFS_c ∪ F_S
9        end
10       for k ∈ [1, K] do
11         S_k ← subsetWithSize(SFS_c, k)
12         for (cand ∈ S_k) ∧ (∄spc ∈ SuspiciousPCSets, cand ⊃ spc) do
13           if satisfy(cand, C_P, C_F) then
14             SuspiciousPCSet.add(cand)
15           end
16         end
17       end
18     end
19     return SuspiciousPCSet
20   end
```

interaction implementation of D includes the statements implementing the interaction of F_E which can be impacted by the features in F_D (if the disabled features were on in p).

In practice, the features in F_D can be mutually exclusive with other features enabled in p, which is constrained by the feature model [1]. Thus, the impact of F_D on the implementation of the interaction of F_E in p, $\beta(F_E, p)$ might not be easily identified via the control/data dependency in p. In this work, the impacts of the features in F_D on statements in p are approximately identified by using *def-use* relationships of the variables and methods that are shared between the features in F_D and p [15]. Formally,

for a statement s, $def(s)$ and $use(s)$ are used to refer to the sets of variables/methods *defined* and *used* by s, respectively.

Definition 4.6 ***(Def-Use Impact).*** *Given an SPL system $\mathfrak{S} = \langle \mathbb{S}, \mathbb{F}, \varphi \rangle$ and its sets of products $\mathbb{P}$, the def-use impact function is defined as γ: $\mathbb{F} \times \mathbb{P} \to 2^{\mathbb{S}}$, where $\gamma(f, p)$ refers to the set of the statements in p which are impacted by any statement in the implementation of feature f, $\varphi(f)$, via the variables/methods shared between $\varphi(f)$ and p. Formally, for a statement s in product p, $s \in \gamma(f, p)$ if one of the following conditions is satisfied:*

- $\exists t \in \varphi(f)$, $def(t) \cap use(s) \neq \emptyset$
- *s is data/control-dependent on s', and $s' \in \gamma(f, p)$*

In summary, in a product $p \in P_F$, for a *Suspicious PC* whose the sets of enabled and disabled features are F_E and F_D, the suspicious statements satisfy the following conditions: **(i)** implementing the interaction of the features in F_E and F_D or impacting this implementation; **(ii)** executing the failed tests of p. For a buggy system, the suspicious space contains all the suspicious statements detected for all *Suspicious PCs* in every failing product of the system.

4.6 Suspicious Statement Ranking

To rank the isolated suspicious statements of a buggy system, VarCop assigns a score to each of these statements based on the program spectra of the sampled products. In VarCop, the suspiciousness of each statement is assessed based on two criteria/dimensions: *Product-based Assessment* and *Test Case-based Assessment*.

4.6.1 Product-based Suspiciousness Assessment

This criterion is based on the overall test results of the products containing the statement. Specially, in a buggy system, a suspicious statement s could be executed in not only the failing products but also the passing products. Hence, from the product-based perspective, *the (dis)appearances of s in the failing and passing products could be taken into account to assess the statement's suspiciousness in the whole system.* In general, the product-based suspiciousness assessment for s could be derived based on the numbers of the passing and failing products where s is contained or not. Intuitively, the more failing products and

the fewer passing products where s is contained, the more suspicious s tend to be. This is similar to the idea of SBFL when considering each product as a test. Thus, this work adopt SBFL metrics to determine the product-based suspiciousness assessment for s. Specially, for a particular SBFL metric M, the value of this assessment is determined by $ps(s, M)$ which adopts the formula of M with the numbers of the passing and failing products containing and not containing s as the numbers of passed and failed tests executed or not executed by s.

4.6.2 Test Case-based Suspiciousness Assessment

The test case-based suspiciousness of a statement s is evaluated based on the detailed results of the tests executed by s. Particularly, in each failing product containing s, the statement is *locally* assessed based on the program spectra of the product. Then, the local scores of s in the failing products are aggregated to form a single value which reflects the test case-based suspiciousness of s in the whole system.

Particularly, the *local test case-based suspiciousness* of statement s can be calculated by using the existing FL techniques such as SBFL [4, 28–31]. This work uses a ranking metric of SBFL, which is the state-of-the-art FL technique, to measure the local test case-based suspiciousness of s in a failing product. Next, for a metric M, the aggregated test case-based suspiciousness of s, $ts(s, M)$, can be calculated based on the local scores of s in all the failing products containing it. In general, one can use any aggregation formula [145], such as *arithmetic mean*, *geometric mean*, *maximum*, *minimum*, and *median* to aggregate the local scores of s.

However, the local test case-based scores of a statement, which are measured in different products, should not be directly aggregated. The reason is that the scores of the statement in different products might be incomparable. Indeed, with some ranking metrics such as Op2 [146] or Dstar [60], once the numbers of tests of the products are different, the local scores of the statements in these products might be in significantly different ranges. Intuitively, if these local scores are directly aggregated, the products which have larger ranges will have more influence on the suspiciousness score of the statement in the whole system. Meanwhile, such larger-score-range products are not necessarily more important in measuring the overall test case-based suspiciousness of the statement. Directly aggregating these incomparable local scores of the statement can result in an inaccurate suspiciousness assessment. Thus, to avoid this problem, these local scores in each product

should be *normalized* into a common scale, e.g., $[0, 1]$, before being aggregated. The impact of the normalization as well as choosing the aggregation function and ranking metric on VarCop's performance will be shown in Section 4.8.2.

4.6.3 Assessment Combination

Finally, the two assessment scores, product-based score, $ps(s, M)$, and test case-based score, $ts(s, M)$, of statement s are combined with a *combination weight*, $w \in [0, 1]$ to form a single suspiciousness score of the statement. Equation 4.1 shows how to combine $ps(s, M)$ and $ts(s, M)$ of s. Note that, to avoid the bias caused by the range-difference between the two criteria, these two scores should be normalized into a common range, e.g., $[0, 1]$ before the interpolation. In the ranking process, the isolated suspicious statements are ranked according to their interpolated suspiciousness score $score(s, M)$. The impact of the combination weight w on VarCop's fault localization performance will be empirically shown in Section 4.8.2.

$$score(s, M) = w * ps(s, M) + (1 - w) * ts(s, M) \tag{4.1}$$

4.7 Empirical Methodology

To evaluate the proposed variability fault localization approach, this chapter seeks to answer the following research questions:

RQ1: ***Accuracy and Comparison.*** How accurate is VarCop in localizing variability bugs? And how is it compared to the state-of-the-art approaches [6, 25, 28]?

RQ2: ***Intrinsic Analysis.*** How do the components including the suspicious statement isolation, the normalization, the suspiciousness aggregation function, and the combination weight contribute to VarCop's performance?

RQ3: ***Sensitivity Analysis.*** How do various factors affect VarCop's performance including the size of sampled product set and the size of test suite in each sampled product?

RQ4: ***Performance in Localizing Multiple Bugs.*** How does VarCop perform on localizing multiple variability bugs?

RQ5: ***Time Complexity.*** What is VarCop's running time?

Table 4.2: Dataset Statistics [5]

System	Details		Test info		Bug info	
	#LOC	*#F*	*#P*	*Cov*	*#SB*	*#MB*
ZipMe	3460	13	25	42.9	55	249
GPL	1944	27	99	99.4	105	267
Elevator-FH-JML	854	6	18	92.9	20	102
ExamDB	513	8	8	99.5	49	214
Email-FH-JML	439	9	27	97.7	36	90
BankAccountTP	143	8	34	99.9	73	310

#F and *#P*: Numbers of features and sampled products.

Cov: Statement coverage (%).

#SB and *#MB*: Numbers of single-bug and multiple-bug cases.

4.7.1 Dataset

To evaluate VarCop, several experiments were conducted on a large public dataset of variability bugs [5]. This dataset includes 1,570 buggy versions with their corresponding tests of six Java SPL systems which are widely used in SPL studies. There are 338 cases of a single-bug, and 1,232 cases of multiple-bug. The details are shown in Table 4.2.

Before running VarCop on the dataset proposed by Ngo et al. [5], a simple inspection was performed for each case on whether the failures of the system are possibly caused by non-variability bugs. Naturally, there are bugs which may be classified as "variability" because of the low-quality test suites which cannot reveal the bugs in some buggy products. There are 53/1,570 cases (19 single-bug cases and 34 multiple-bug cases) where among the sampled products in each case, the product containing only the base feature and disabling all of the optional features fails several tests. These cases possibly contain non-variability bugs, this problem will be discussed them in Section 4.8.2.

4.7.2 Evaluation Setup, Procedure, and Metrics

Empirical procedure

Comparative study. For each buggy version, this work compared the performance in ranking buggy statements of VarCop, Arrieta et al. [6], SBFL [4, 28–31], and the combination of slicing method and SBFL (*S-SBFL*) [120, 121] accordingly. For SBFL, each SPL system is considered as a non-configurable code. SBFL ranks all the statements executed by failed tests. For S-SBFL, to improve SBFL, S-SBFL isolates all the executed failure-related statements in every failing product by slicing techniques [26] before ranking. A failure-related statement is a statement included in at least a program slice which is backward sliced from a failure point in a failing product. In this experiment, 30 most popular SBFL metrics [4, 28, 29] were used. For each metric, M, this experiment compared the performance of all four techniques, including VarCop, SBFL, S-SBFL, and Arrieta et al. [6], using M.

Intrinsic analysis. This experiment studied the impacts of the following components: *Suspicious Statement Isolation*, *Ranking Metric*, *Normalization*, *Aggregation Function*, and *Combination Weight*. Different variants of VarCop with different combinations were created and then their performance is measured accordingly.

Sensitivity analysis. This experiment studied the impacts of several factors VarCop's performance: *Sample size* and *Test set size*. To systematically vary these factors, the sample size is varied based on k-wise coverage [53] and the tests are gradually added.

Metrics

Rank, *EXAM* [140], and *Hit@X* [28, 147] which are widely used in evaluating FL techniques [4, 6, 28] were adopt. For the cases of multiple variability bugs, *Proportion of Bugs Localized (PBL)* [28] are additionally applied for measure FL performance.

Rank. Rank indicates the position of the buggy statements in the resulted lists of the FL techniques. The lower rank of buggy statements, the more effective approach. If there are multiple statements having the same score, buggy statements are ranked last among them. Moreover, for the cases of multiple bugs, this work measured *Rank*s of the first buggy statement (*best rank*) in the lists.

EXAM. *EXAM* [140] is the proportion of the statements being examined until the first

faulty statement is reached, Equation 4.2. In this Equation, r is the position of the buggy statement in the ranked list and N is the total number of statements in the list. The lower *EXAM*, the better FL technique.

$$EXAM = \frac{r}{N} \times 100\% \tag{4.2}$$

Hit@X. *Hit@X* [28, 147] counts the number of bugs which can be found after investing X ranked statements, e.g., *Hit@1* counts the number of buggy statements correctly ranked 1st among the experimental cases. In practice, developers only investigate a small number of ranked statements before giving up[148]. Thus, this work focuses on $X \in [1, 5]$.

Proportion of Bugs Localized (PBL). *PBL* [28] is the proportion of the bugs detected after examining a certain number of the statements. The higher *PBL*, the better approach.

4.8 Empirical Results

4.8.1 Accuracy and Comparison (RQ1)

Table 4.3 shows the average performance of VarCop, SBFL, the combination of slicing method and SBFL (**S-SBFL**), and the feature-based approach proposed by Arrieta et al. [6] (**FB**) on 338 buggy versions containing a single bug each [5] in *Rank* and *EXAM*.

Compare to SBFL and S-SBFL. *For both Rank and EXAM,* VarCop *outperformed S-SBFL and SBFL in **all** the studied metrics.* On average, VarCop achieved **33%** better in *Rank* compared to S-SBFL and nearly **50%** compared to SBFL. This means, to find a variability bug, by using VarCop, developers have to investigate only 5 statements instead of about 8 and up to 10 suspicious statements by S-SBFL and SBFL. In *EXAM*, the improvements of VarCop compared to both S-SBFL and SBFL are also significant, **30%** and **43%**, respectively. For developer, the proportion of statements they have to examine is reduced by about one third and one haft by using VarCop compared to S-SBFL and SBFL. For the 5 most popular metrics [4], [59] (in shade) including Tarantula [17], Ochiai [58], Op2 [146], Dstar [60], and Barinel [30], VarCop achieved the improvement of more than **15%**. Especially, for certain metrics such as Simple Matching [149], the improvements by VarCop are remarkable, **4 times** and **35 times** compared to S-SBFL and SBFL.

Table 4.3: Performance of VarCop, SBFL, the combination of Slicing and SBFL (*S-SBFL*), and Arrieta et al. [6] (*FB*)

#	Ranking Metric	***Rank***				***EXAM***			
		VarCop	**S-SBFL**	**SBFL**	**FB**	**VarCop**	**S-SBFL**	**SBFL**	**FB**
1	Barinel	7.83	9.88	11.48	136.27	2.11	2.87	3.15	21.79
2	Dstar	6.16	7.20	8.09	108.78	1.77	1.94	2.02	15.88
3	Ochiai	6.19	7.25	8.14	109.91	1.77	1.95	2.03	16.10
4	Op2	5.86	6.07	6.74	106.99	1.71	1.75	1.80	15.36
5	Tarantula	6.96	9.88	11.48	136.27	1.98	2.87	3.15	21.79
6	Kulczynski2	5.61	6.36	7.08	108.23	1.67	1.77	1.83	15.59
7	M2	5.94	6.13	6.82	108.43	1.71	1.76	1.81	15.77
8	Harmonic Mean	5.95	6.52	7.28	149.70	1.72	1.80	1.86	21.37
9	Zoltar	6.00	6.12	6.78	107.57	1.68	1.75	1.80	15.45
10	Geometric Mean	6.05	7.37	8.29	149.70	1.76	1.99	2.09	21.37
11	Ample2	6.15	6.16	6.86	149.58	1.75	1.77	1.82	21.30
12	Rogot2	6.22	6.52	7.28	133.66	1.80	1.80	1.86	22.24
13	Sorensen Dice	6.50	8.79	10.17	115.72	1.84	2.41	2.62	17.29
14	Goodman	6.50	8.79	10.17	115.72	1.84	2.41	2.62	17.29
15	Jaccard	6.63	8.79	10.17	115.72	1.83	2.41	2.62	17.29
16	Dice	6.63	8.79	10.17	115.72	1.83	2.41	2.62	17.29
17	Anderberg	6.68	8.79	10.17	115.72	1.84	2.41	2.62	17.29
18	Cohen	6.81	8.93	10.33	152.04	1.87	2.47	2.70	21.61
19	Fleiss	6.82	12.24	52.03	145.70	2.09	3.51	9.13	21.65
20	Simple Matching	6.88	28.00	242.70	158.19	2.11	6.67	30.68	21.96
21	Humman	6.88	28.00	242.70	158.19	2.11	6.67	30.68	21.96
22	Wong2	6.88	28.00	242.70	158.19	2.11	6.67	30.68	21.96
23	Hamming	6.88	28.00	242.70	158.19	2.11	6.67	30.68	21.96
24	Sokal	6.91	28.00	242.70	158.19	2.15	6.67	30.68	21.96
25	Euclid	6.96	28.00	242.70	158.19	2.17	6.67	30.68	21.96
26	Rogers Tanimoto	7.05	28.00	242.70	158.19	2.20	6.67	30.68	21.96
27	Scott	7.38	13.22	50.86	147.65	2.17	3.76	8.79	22.23
28	Rogot1	7.38	13.22	50.86	147.65	2.17	3.76	8.79	22.23
29	Russell Rao	14.06	17.87	24.00	309.62	3.58	5.05	6.39	39.27
30	Wong1	14.06	17.87	24.00	309.62	3.58	5.05	6.39	39.27

Listing 4.2: A variability bug in system *ExamDB*

```
1  public boolean consistent(){
2      for (int i = 0; i < students.length; i++) {
3          if (students[++i] != null && !students[i].backedOut && students[i].points < 0) {
4              //Patch:  students[i] != null
5              return false;
6          }
7      }
8      return true;
9  }
```

There are two reasons for these improvements. *Firstly, the set of the suspicious statements isolated by* VARCOP *is much smaller than other approaches'*. VARCOP identifies the suspicious statements by analyzing the root causes of failures. The suspicious space by VARCOP is only about 70% of the space of S-SBFL and 10% of the space of SBFL. The average suspicious space isolated by VARCOP is only 66 statements, while the isolated set by S-SBFL contains 87 statements, and this suspicious set identified by SBFL is even much larger, 660 statements. *Secondly, the suspiciousness of statements computed by* VARCOP *is not biased by the tests in any specific product.* Unlike SBFL, in VARCOP, for a suspicious statement s, the appearances of s in both passing and failing products, as well as the local test case-based suspiciousness scores of s in all the failing products containing it are aggregated appropriately. This suspiciousness measurement approach helps VARCOP overcome the weakness of SBFL in computing suspiciousness for the statements of systems.

Listing 4.2 shows a variability bug (ID_298) in feature *BackOut* of *ExamDB*. In this code, each member in `students` must be visited. However, the students with the even index are incorrectly ignored because `i` is increased twice after each iteration (line 2 and line 3). This bug is revealed only when both *ExamDB* and *BackOut* are enabled. For QA, 8 products are sampled for testing. There are 4 failing products and total 168 statements executed during running failed tests. By using Tarantula [17], VARCOP ranked the buggy statement [4] s_3 (line 3) first, while SBFL ranked it 10th. Indeed by locally ranking the buggy statement in each product, s_3 is ranked 1st in 3 out of 4 failing products. Meanwhile, this statement is ranked at the 27th position in the ranked list of the other failing

[4]Note that Bugs could have locality property. For a failure, developers could modify different statements to repair the system. In the example in Listing 4.2, the bug could be in s_2 or s_3. For ease of evaluation and comparison, in the experiments, the buggy statements are statements marked "incorrect" and repaired by the benchmark. Thus, s_3 is considered as buggy statements (instead of s_2) in this example.

product (p_4). In p_4, there are 10 correct statements which are executed by failed tests only, yet not executed by any passed test. By using Tarantula, SBFL assigned these statements the highest suspiciousness scores in p_4. Thus, those statements have higher scores than the buggy statement which is executed during running both the failed and passed tests. In the whole system, SBFL uses the test results of all the sampled products to measure the suspiciousness for all the 168 statements. Consequently, it misleadingly assigned higher scores for all of the 10 statements which are executed by only the failed tests in p_4, yet not executed by any tests in the others. Consequently, the ranking result by SBFL is considerably driven by the test results of p_4 and mislocates the buggy statement. Meanwhile, VarCop ignored 101/168 failure-unrelated statements and measured the suspiciousness of only 67 statements by analyzing the root cause of the failures. Additionally, in VarCop, the test case-based suspiciousness of these statements are aggregated from the suspiciousness values which are measured in the failing products independently. Thus, the low-quality test suite of p_4 cannot significantly affect the suspiciousness measurement, and the buggy statement is still ranked 1st thanks to the test suites of other products.

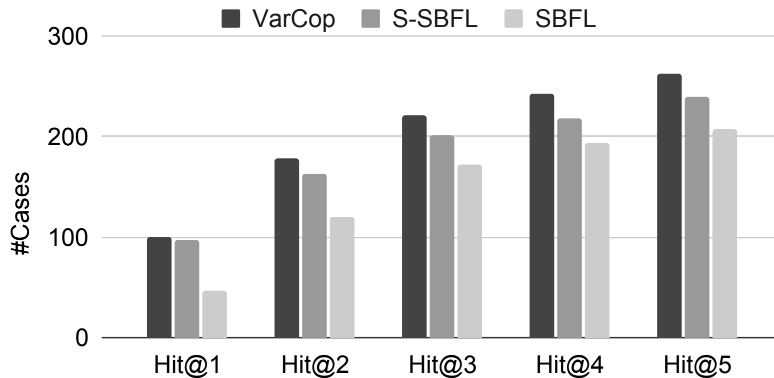


Figure 4.2: *Hit@1–Hit@5* of VarCop, S-SBFL and SBFL

Furthermore, VarCop *also surpasses S-SBFL and SBFL in Hit@X*. In Figure 4.2, after investigating X statements, *for* $X \in [1, 5]$*, there are more bugs found by using* VarCop *compared to S-SBFL and SBFL*. On average, in **78%** of the cases, VarCop correctly ranked the buggy statements at the top-5 positions, while S-SBFL and SBFL ranked them at the top-5 positions in only 70% and 61% of the cases, respectively. Moreover, in about two-thirds of the cases (**+65%**), the bug can be found by examining **only first 3 statements** in the lists of VarCop. Meanwhile, to cover the same proportion of the cases by using S-SBFL and SBFL, developers have to investigate up to 4 and 5 statements.

Table 4.4: Performance by Mutation Operators

Group	**Mutation Operator**	**#Bugs**	***Rank***	***EXAM***
Conditional	*COR, COI, COD*	32	1.63	0.61
Assignment	*ASRS*	7	2.14	0.89
Logical	*LOI*	17	2.47	2.21
Deletion	*CDL, ODL*	18	3.56	1.09
Relational	*ROR*	52	5.13	1.29
Arithmetic	*AODU, AOIU, AORB, AOIS, AORS, AODS*	212	7.63	2.01

Especially, for *Hit@1*, the number of bugs are found by VarCop after investigating the first ranked statements is about **101 bugs (30%)**. This means, in **one-third of the cases**, developers just need to examine the first statements in the ranked lists to find bugs by using VarCop.

Compare to Arrieta et al.[6]. As illustrated in Table 4.3, in all the studied metrics, VarCop outperformed Arrieta et al. [6] **21 times** in *Rank* and **11 times** in *EXAM*. Instead of ranking statements, this approach localizes the variability bugs at the feature-level. Consequently, all the statements in the same feature are assigned to the same score. In Listing 4.2, the buggy statement at line 3 is assigned the same suspiciousness level with 22 correct statements. Thus, even the feature containing the fault, *BackOut*, is ranked first, the buggy statement is still ranked 22nd in the statement-level fault localizing. Unfortunately, *BackOut* is actually ranked 4th, then the buggy statement is ranked 87th. This could lead to the ineffectiveness of the feature-based approach proposed by Arrieta et al.[6] in localizing variability bugs in the statement-level.

Overall, the results show that VarCop *significantly outperformed the state-of-the-art approaches, S-SBFL, SBFL, and Arrieta et al. [6], in all 30/30 SBFL ranking metrics.*

Performance by bug types

This experiment further analyzed VarCop's performance on localizing bugs in different types based on mutation operators [150] and kinds of code elements [151].

Table 4.5: Performance by Code Elements of Bugs

Code Element	**#Bugs**	***Rank***	***EXAM***
Method Call	22	4.23	0.37
Conditional	148	5.20	1.86
Loop	17	6.41	2.27
Assignment	108	7.18	1.82
Return	43	7.21	1.33

In Table 4.4, VarCop performs most effectively on the bugs created by *Conditional Operators* which are ranked between 1st and 2nd on average. The reason is that these bugs are easier to be detected (killed) by the original tests than other kinds of mutants [152]. This means, if the bugs in this kind can cause the failures in some products, the bugs will be easier to be revealed by the products' tests. Moreover, the correct states of either passing or failing of products affect the performance of FL techniques. As a result, this kind of bug is more effectively localized by VarCop. Meanwhile, VarCop did not localize well the bugs created by *Arithmetic Operators*, as they are more challenging to be detected by the original tests set of the products [152]. Indeed, because of the ineffectiveness of the test suites in several products, even they contain the bug(s), their test suites cannot detect the bugs, and the products still pass all their tests. In these cases, the performance would be negatively affected.

Table 4.5 shows VarCop's performance in different kinds of bugs categorized based on code elements [151]. As seen, VarCop *works quite stably in these kinds of bugs*. Particularly, the average *Rank* achieved by VarCop for bugs in different code elements is between 4th and 7th, with the standard deviation is only 1.3. In addition, the average $EXAM$ and the standard deviation are 1.53 and 0.73, respectively.

Performance by the number of involving features

This experiment analyzed the performance of VarCop by the number of the features which involve in the visibility of the bugs [5]. In the experiment, the number of involving features is in the range of $[1, 25]$. In +76% of the cases, the number of involving features is fewer than or equal to 7. VarCop's performance in *Rank* by the numbers of involving

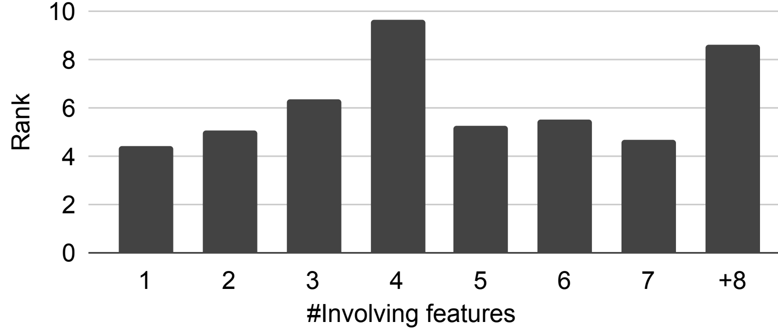


Figure 4.3: Performance by number of involving features of bugs

features are not significantly different, from 4.45 to 9.69, Figure 4.3. In fact, both the number of detected *Suspicious PCs* and the size of each *Suspicious PC*, which determine the size of isolated suspicious space, are affected by the number of involving features. Specially, for the bugs with a smaller number of involving features, the detected *Suspicious PCs* are likely fewer, but each of them is likely smaller. With a larger number of involving features, the detected *Suspicious PCs* tend to be more, yet each of these *Suspicious PCs* is likely larger. For a bug, the isolated suspicious statements space is in direct proportion to the number of detected *Suspicious PCs*, but it is in inverse proportion to the size of each *Suspicious PC*. Thus, the number of involving features would not linearly affect the number of isolated suspicious statements and VarCop's performance.

4.8.2 Intrinsic Analysis (RQ2)

Impact of suspicious statements isolation on performance

To study the impact of Suspicious Statements Isolation (Figure 4.1), which includes *Buggy PC* Detection and Suspicious Statements Identification components, on VarCop's performance, this experiment built the variant of VarCop where these two components are disabled. For a buggy system, this variant of VarCop ranks all the statements which are executed during running failed tests in the failing products. Figure 4.4 shows the performance of VarCop using 5 most popular SBFL metrics [4], [59] when *Buggy PC* Detection and Suspicious Statements Identification are enabled/disabled. As expected, *when enabling these components, the performance of* VarCop *is significantly better, about 16% in Rank.*

Interestingly, *even when disabling Suspicious Statements Isolation, this variant of* Var-

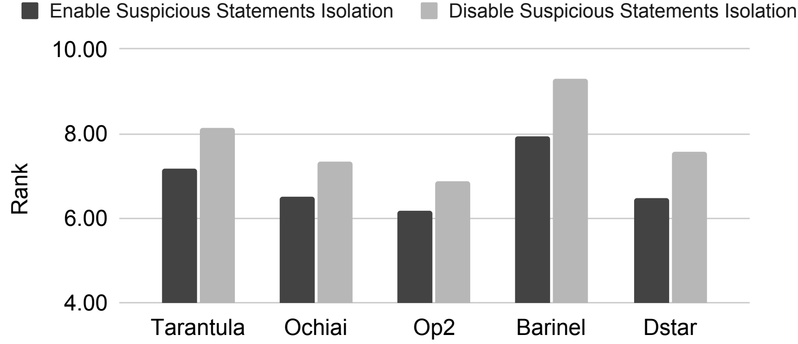


Figure 4.4: Impact of *Buggy PC* Detection on performance

Cop *is still better than S-SBFL and SBFL*. Specially, VarCop obtained a better *Rank* than S-SBFL and SBFL in 21/30 and 27/30 metrics, respectively. In these ranking metrics, the average improvements of VarCop compared to S-SBFL and SBFL are 34% and 45%. Meanwhile, for the remaining metrics, the performances of S-SBFL and SBFL are better than VarCop by only 10% and 3%. For example, this variant of VarCop ranked the buggy statement (s_3) in Listing 4.2 at 1st which is much better than the 9th and 10th positions by S-SBFL and SBFL.

Note that, for 19/338 cases which possibly contain non-variability bugs (mentioned in Section 4.7.1), there might be no *Buggy PC* in these buggy systems to be detected. Moreover, the low-quality test suites in some passing (yet buggy) products might "fool" fault localization techniques [100]. These passing products might also make VarCop less effective in isolating suspicious statements. Hence, for these cases, turning off VarCop's suspicious statements isolation component helps to guarantee its effectiveness.

Impact of ranking metric on performance

This experiment studied the impact of the selection of the local ranking metric on VarCop's performance. To do that, different variants of VarCop with different metrics were built. In Table 4.3 (the 3rd and 7th columns), *the performance of* VarCop *is quite stable with the different ranking metrics*. Particularly, for all the studied metrics, the average *EXAM* achieved by VarCop is in a narrow range, from 1.71–3.58, with the standard deviation of 0.46. Additionally, the average *Rank* of the buggy statements assigned by VarCop is varied from 6th–14th. This stability of VarCop is obtained due to the suspicious statements isolation and suspiciousness measurement components. Indeed, VarCop

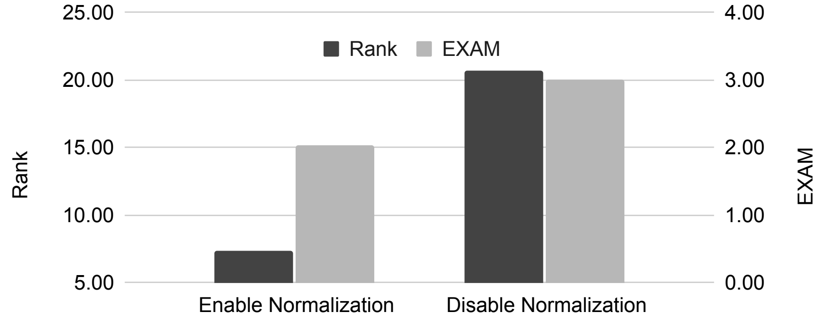


Figure 4.5: Impact of Normalization on performance

only considers the statements that are related to the interactions which are the root causes of the failures. Moreover, VarCop is not biased by the test suites of any specific products. Consequently, its performance is less affected by the low-quality test suites of any product. Thus, *selecting an inappropriate ranking metric, which is unknown beforehand in practice, does not significantly affect* VarCop*'s performance*. This demonstrates that VarCop is practical in localizing variability bugs.

In contrast, the performances of S-SBFL and SBFL techniques are considerably impacted by choosing the ranking metrics. By S-SBFL method, the average *Rank* of the buggy statements widely fluctuates from 6th to 28th. By SBFL, the fluctuation of *Rank* is even much more considerable, from 7th to 243rd. Consequently, the QA process would be extremely inefficient if developers using the SBFL technique with an inappropriate ranking metric.

Impact of normalization on performance

To study the impact of the normalization, this experiment built the variants of VarCop which enable and disable the normalization component. In this experiment, in both cases, the local test case-based scores are accordingly measured by 30 popular SBFL metrics and are aggregated by *arithmetic mean*.

In Figure 4.5, when enabling the *normalization*, VarCop's performance is better than when *normalization* is off. Particularly, the performance of VarCop is improved 64% in *Rank* and 32% in *EXAM* when the *normalization* is enabled. One reason is that for some SBFL metrics, such as Fleiss [153] and Humman [154], the ranges of the product-based and test case-based suspiciousness values are *significantly different*. Additionally, for these

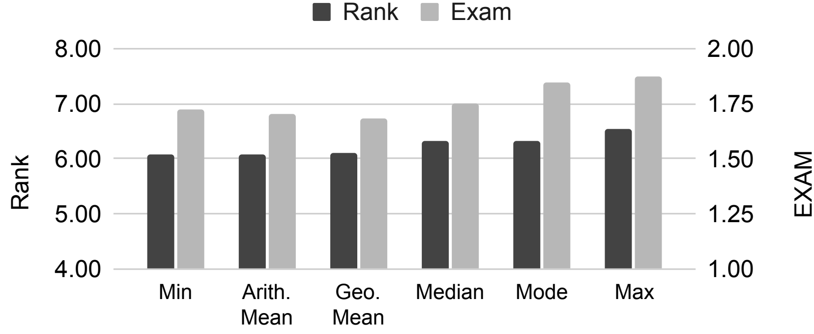


Figure 4.6: Impact of choosing $score(s, M)$ on performance

metrics, the ranges of the local test case-based suspiciousness scores in different products are also *significantly different*. For example, there is a bug (ID_25) in the system *Email*, with Fleiss, the range of suspiciousness scores in product p_1 is $[-33, 1.8]$, while the range in another product, p_2 is much different, $[-8.6, 1.78]$. Without *normalization*, a statement in p_2 is more likely to be assigned a higher final score than one in p_1. Meanwhile, with several metrics such as Ochiai [58] and Tarantula [17], the performance of VarCop is slightly different when *normalization* is on/off. For these metrics, the local scores of the statements in the products are originally assigned in quite similar ranges. Thus, these local scores might not need to be additionally normalized. *Overall, to ensure that the best performance of* VarCop*, the normalization should be on.*

Impact of aggregation function on performance

To study the impact of choosing aggregation function on performance, this experiment varied the aggregation function of the test case-based suspiciousness assessment. In this experiment, Op2 [146] was randomly chosen to measure the local scores of the statements. As seen in Figure 4.6, *the performance of* VarCop *is not significantly affected when the aggregation function is changed.* Specially, the average *Rank* of VarCop is around $6^{th} - 7^{th}$, while the *EXAM* is about 1.76.

Impact of combination weight on performance

This experiment varied the *combination weight* $w \in [0, 1]$ (Section 4.6) when combining the product-based and test case-based suspiciousness assessments to form the final score of statements. Figure 4.7 shows the average *Rank* and *EXAM* of the faults in 36 buggy

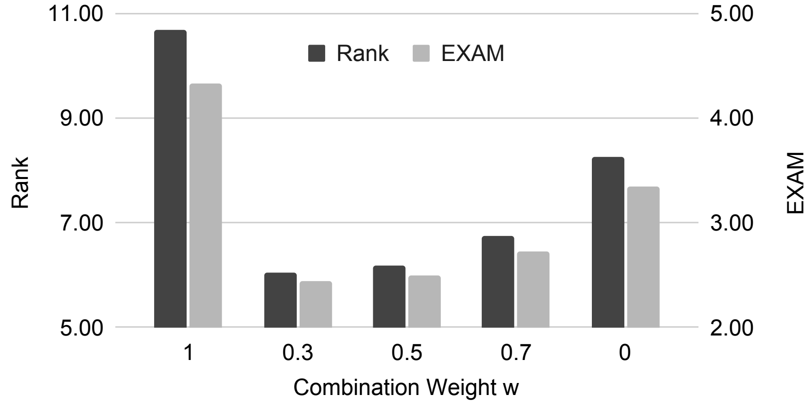


Figure 4.7: Impact of choosing combination weight on performance

versions of *Email*, with $w \in [0, 1]$.

As seen, *the performance of* VarCop *is better when both the product-based and test case-based suspiciousness scores are combined to measure the suspiciousness of the statements.* For $w = 1$, the statements are ranked by only their product-based suspiciousness. All the statements in the same feature will have the same suspiciousness score because they appear in the same number of passing and failing products. For $w = 0$, a statement is ranked by only the score which is aggregated from the local scores of the statement in the failing products. Consequently, the overall performance may be affected by the low-quality test suites of some products. For instance, the correct statement s appears in only one failing product p. However, in p, s is misleadingly assigned the highest score. As a result, when $w = 0$, s also has the highest score in the whole system, since this score is aggregated from p, the only failing product containing s. Hence, *both of the product-based and test case-based suspiciousness assessments are necessary for measuring the suspiciousness of statements* (Mentioned in **O3**).

4.8.3 Sensitivity Analysis (RQ3)

Impact of sample size on performance

For each buggy version of *GPL* which is randomly selected system, this experiment used k-wise coverage, for $k \in [1, 4]$, to systematically vary sample size. Then, VarCop was experimented on each case with each set of sampled products.

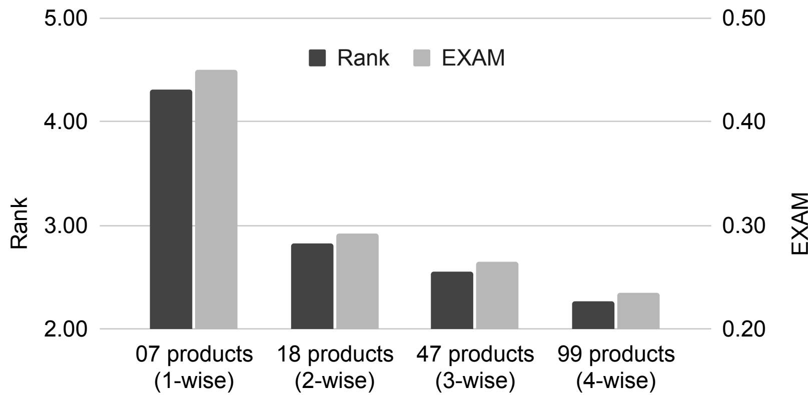


Figure 4.8: Impact of the sample size on performance

Figure 4.8 shows the average *Rank* and *EXAM* of VarCop in the buggy versions of *GPL* with different sample sets. As expected, *the larger sample, the higher performance in localizing bugs obtained by* VarCop. However, when the ranking results reach a specific point, even more products are tested, the results are just slightly better. Specially, for the set of 1-*wise* coverage (*One-disabled* [14]), the average *Rank* and *EXAM* are about 4.31 and 0.45, respectively. Meanwhile, for 2-*wise*, the ranking results are 1.5 times better. The reason is that, for a case, the more products are tested, the more information VarCop has to detect *Buggy PCs* and rank suspicious statements. However, compared to *3-wise* and *4-wise*, even much more products are sampled, which is much more costly in sampling and testing, the performance is just slightly improved. Hence, *with* VarCop*, one might not need to use a very large sample to achieve a relatively good performance in localizing variability bugs.*

Impact of test suite's size on performance

For every buggy version, this experiment gradually increased the size of the test suite in each product to study the impact of tests on VarCop's performance. The randomly selected system in this experiment is *ExamDB*.

In Figure 4.9, VarCop*'s performance is improved when increasing the test suite size.* Particularly, when the number of tests increased from 13 to 90 tests/product, both *Rank* and *EXAM* of VarCop are improved by about twice. After that, even more tests are added, VarCop's performance is just slightly affected. The reason is, increasing the number

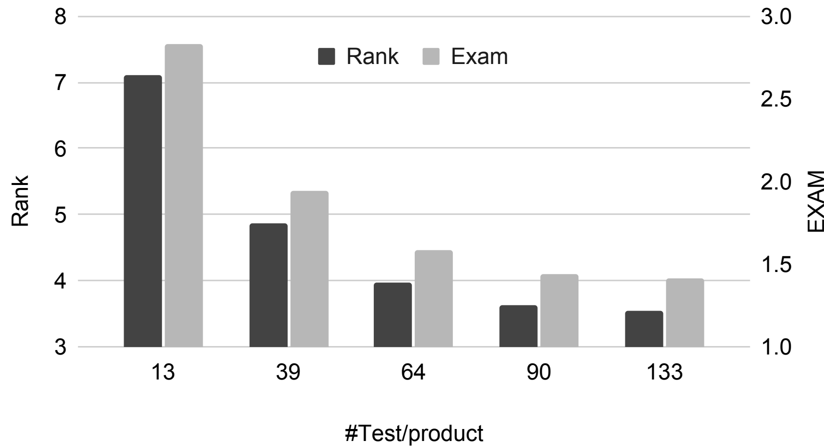


Figure 4.9: Impact of the size of test set on performance

of tests provides more information to distinct the correct and incorrect statements, thus improves VarCop's performance. However, when the test suites reach a specific effectiveness degree in detecting bugs, added tests would not provide supplementary information for the FL process.

Overall, *one should trade off between the fault localization effectiveness and the cost of generating tests and running them. Furthermore, as discussed in Section 4.8.2, instead of focusing on expanding test suites for products, developers should improve the effectiveness of the test suites (e.g., test coverage) in detecting bugs.*

4.8.4 Performance in Localizing Multiple Bugs (RQ4)

To evaluate VarCop on buggy systems that contain multiple variability bugs, the dissertation conducted an experiment on 1,232 buggy versions of the subject systems with 2,947 variability bugs in total. The components of VarCop were randomly configured: the ranking metric is Op2 [146], and the aggregation function is *arithmetic mean.*

Figure 4.10 shows that the average percentage of buggy statements in each case found (PBL) by VarCop far surpasses the corresponding figures of S-SBFL and SBFL when the same number of statements are examined in their ranked lists. Specially, after examining the first statement, VarCop can find nearly **10%** of the incorrect statements in a buggy system. Meanwhile, only 5% and 1% of the bugs are found by S-SBFL and SBFL, respectively, after inspecting the first statement. Furthermore, about **35%** of the bugs can be found by VarCop by checking only first **5** statements in the ranked lists. Meanwhile,

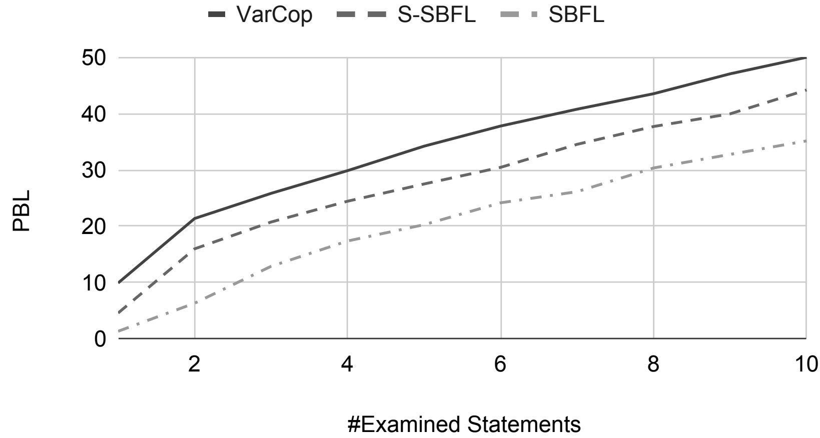


Figure 4.10: VarCop, S-SBFL and SBFL in localizing multiple bugs

with S-SBFL and SBFL, developers have to investigate up to 7 and even 10 statements to achieve the same performance. In addition, the average *best Rank* that VarCop assigned for the buggy statements is about 7th. Meanwhile, the corresponding figures of S-SBFL, SBFL, and Arrieta et al. [6] are 7.5^{th}, 10th, and 293th, respectively.

Especially, in **22%** of the cases, VarCop correctly ranked at least one of the bugs at the **top-1 positions**, while for S-SBFL, SBFL, and Arrieta et al. [6], the corresponding proportions are only 10%, 3%, and 0%. In the experiment, at least one of the buggy statements of about **65%** of cases are correctly ranked at top-5 positions by VarCop.

Listing 4.3: A variability bug in feature *Number*

```
1 public void preVisitAction(Vertex v){
2    if (v.visited != true) {
3       v.VertexNumber = --vertexCounter;
4       //Patch:  v.VertexNumber = vertexCounter++;
5    }
6 }
```

Listing 4.3 and Listing 4.4 show a case (ID_143) containing two buggy statements in *GPL*. For the bug in Listing 4.3, VarCop correctly ranked it 1st. Since in practice, the number of incorrect statements is unknown beforehand, developers have to conduct the testing and debugging processes in regression. After fixing the first bug, developers have to perform regression testing. One more time, VarCop effectively ranked the bug in Listing 4.4 at the 2nd position. Thus, developers can continuously use VarCop in the regression process to quickly find all the bugs in the systems. Meanwhile, by using SBFL,

two buggy statements are ranked 4th and 5th, respectively. This shows *the effectiveness of* VarCop *in localizing multiple variability bugs in SPL systems.*

Listing 4.4: Another variability bug in *WeightedWithNeighbors*

```
1  public void addEdge( Vertex start, Neighbor theNeighbor){
2    original( start, theNeighbor );
3    if (isDirected != false) {
4      //Patch:  isDirected == false
5      Vertex end = theNeighbor.neighbor;
6      end.addWeight( end, theNeighbor.weight );
7    }
8  }
```

4.8.5 Time Complexity (RQ5)

The experiments were conducted on a desktop with Intel Core i5 2.7GHz, 8GB RAM. In 65% of the cases, VarCop took only about 2 minutes to automatically localize buggy statements in each case. On average, VarCop spent about 20 minutes on a buggy SPL system. Furthermore, this chapter studies VarCop's running time in different input aspects including the sample size and the complexity of buggy systems in LOCs.

Particularly, the running time of VarCop gracefully increased when more products were used to localize bugs (no show). This is expected because to localize the variability bugs, VarCop needs to examine the configurations of the more sampled products to detect *Buggy PCs*. Additionally, VarCop analyzes the detected *Buggy PCs* in these products to isolate the suspicious statements. Then calculating their suspiciousness scores, as well as ranking these statements.

For the complexity of buggy systems, in general, VarCop took more time to analyze the system which has more lines of code (no show). In particular, VarCop took the least time to localize bugs in the smallest system, *BankAccountTP*, while it took more time to investigate a case in the larger one, *ZipMe*. However, VarCop needs more time to investigate a case in *Email* compared to a case in *ExamDB*, which is larger than *Email* in terms of LOC. The reason is, VarCop analyzes all the failing products to isolate suspicious statements. In the experiment, there are many cases where the number of failing products in *Email* is larger than the number of failing products in *ExamDB*.

4.8.6 Threats to Validity

The main threats to the validity of this work are consisted of three parts: internal, construct, and external validity threat.

Threats to internal validity mainly lie in the correctness of the implementation of the proposed approach. To reduce this threat, the code was manually reviewed and the program analysis tools' outputs were carefully verified.

Threats to construct validity mainly lie in the rationality of the assessment metrics. To reduce this threat, the metrics, which have been recommended by prior studies/surveys [29] and widely used in previous work [4, 28], were chosen.

Threats to external validity mainly lie in the benchmark used in the experiments. The artificial bugs are generated by the mutation testing tool, this will make the diversity of artificial faults in the benchmark, yet could also introduce a bias in the evaluation. To reduce this threat, the experiments are conducted on various kinds of mutation operators on both single-bug and multiple-bug settings. In addition, there is also a threat to external validity is that the obtained results on artificial faults can not be generalized for large-scale SPL systems containing real faults. To mitigate the threat, six systems, which were widely-used in existing studies [13, 16, 155], were chosen. These systems target different application domains. VarCop obtained consistent results on these systems. Moreover, although it has been very common to evaluate and compare fault localization techniques using artificial faults as a proxy to real faults [4], it remains an open question whether results on artificial faults are characteristic of results on real faults. In future work, I am planning to create manually-seeded faults and collect more real-world variability bugs in larger SPL systems to evaluate the proposed technique to address these threats. As another external threat, all systems in the benchmark are developed in Java. Therefore, it cannot claim that similar results would have been observed in other programming languages or technologies. This is a common threat of several studies on configurable software systems [83, 156]. Another threat is that the selected SBFL metrics might not be representative. To reduce the threat, a large number of the most popular SBFL metrics [4, 28, 29] were chosen.

4.9 Summary

This chapter introduces VarCop, a novel approach for localizing variability bugs. First, to isolate the suspicious statements, VarCop analyzes the overall test results and the failing products' code to detect the statements related to the interactions that potentially make the bugs (in)visible in the products. Then, VarCop ranks each isolated statement based on two suspiciousness dimensions which are measured by both the overall test results of the products and the detailed results of the test cases which are executed by the statement. Several experiments were conducted on a large dataset of buggy versions of 6 SPL systems in both single-bug and multiple-bug settings. The experimental results show that in the all 30/30 most popular ranking metrics, VarCop's performance in *Rank* is 33%, 50% and 95% better than the state-of-the-art FL techniques, S-SBFL, SBFL, and Arrieta et al. [6], for single-bug cases. Moreover, VarCop correctly ranked the buggy statement in +65% of the cases at the top-3 positions in the resulting lists. For the cases of multiple-bug, one-third of the bugs in a buggy system are ranked at top-5 positions by VarCop. Especially, in 22% and 65% of the cases, VarCop is able to effectively localize at least one buggy statement at top-1 and top-5 positions of its ranked lists, respectively. From that, developers can iterate the process of bugs detecting, bugs fixing, and regression testing to quickly fix all the bugs and assure the quality of SPL systems.

This work was published in the IEEE Transactions on Software Engineering in 2021. Nguyen, Thu-Trang, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. "A variability fault localization approach for software product lines." IEEE Transactions on Software Engineering 48, no. 10 (2021): 4100-4118. (ISI/Q1)

Chapter 5

Automated Variability Fault Repair

This chapter proposes two approaches, *product-based* and *system-based*, to automatically repair the variability bugs in an SPL system to fix the failures of the failing products and not to break the correct behaviors of the passing products. Firstly, this chapter introduces the concepts of APR and SPL, as well as the problem of repairing variability bugs in SPL systems. Next, this chapter presents the basic product-based and system-based approach ($ProdBased_{basic}$ and $SysBased_{basic}$) for repairing variability bugs. Then, this chapter introduces the enhanced variants of these approaches ($ProdBased_{enhanced}$ and $SysBased_{enhanced}$), which embed the heuristic rules.

5.1 Introduction

In practice, bugs are an inevitable problem in software programs. Developers often need to spend about 50% of their time on bug addressing [34]. Detecting and fixing bugs in SPL systems could be very complicated due to their variability characteristics. Echeverría et al. [35] conducted an empirical study to evaluate engineers' behaviors in fixing errors and propagating the fixes to other products in an industrial SPL system. They showed that fixing buggy SPL systems is challenging, especially for large systems. Indeed, in an SPL system, each product is composed of a different set of features. Due to the interaction of different features, a bug in an SPL system could manifest itself in some products of the system but not in others, so called *variability bugs*. In order to fix variability bugs, the APR tools need to find patches which not only work for one product but also for all the products of the system, i.e., these tools need to fix the incorrect behaviors of all *failing products*, and do not break the correct behaviors of the *passing products*.

To reduce the cost of software maintenance and alleviate the heavy burden of manually debugging activities, multiple automatic program repair (APR) approaches [18, 36–40] have been proposed in recent decades. These approaches employ different techniques to automatically (i.e., without human intervention) synthesize patches that eliminate program faults and obtain promising results. However, these approaches focus on fixing bugs in a single non-configurable system.

In the context of SPL systems, there are several studies attempting to deal with the variability bugs at different levels, such as model or configuration. For example, Arcaini et al. [41, 42] attempt to fix bugs in the variability models. Weiss et al. [43, 44] repair misconfigurations of the SPL systems. However, repairing variability bugs at the source code level still remains unexplored.

This research aims to make the first attempt at automatically repairing variability bugs in the source code of SPL systems. This chapter proposes two approaches, *product-based* and *system-based*, for repairing buggy SPL systems at the source code level. For the *product-based approach* ($ProdBased_{basic}$), each failing product of the system is repaired individually, and then the obtained patches, which cause the product under repair to pass all its tests, are propagated and validated on the other products of the system. For the *system-based approach* ($SysBased_{basic}$), instead of repairing one individual product at a time, all the products are considered for repairing simultaneously. Specifically, the patches are generated and then validated by all the sampled products of the system in each repair iteration. For both approaches, the valid patches are the patches causing all the available tests of all the sampled products of the system to pass.

Furthermore, this chapter also introduces several *heuristic rules* for improving the performance of the two approaches in repairing buggy SPL systems. The heuristic rules are started from the observation that, in order to effectively and efficiently fix a bug, an APR tool must correctly decide (i) where to fix (*navigating modification points*) and (ii) how to fix (*selecting suitable modifications*). The heuristic rules focus on enhancing the accuracy of these tasks by leveraging intermediate validation results of the repair process.

For *navigating modification points*, APR tools [38, 48] often utilize the *suspiciousness scores*, which refer to the probability of the code elements to be faulty. These scores are often calculated once for all before the repair process by FL techniques such as spectrum-based [25, 31] or mutation-based FL [157]. However, a lot of additional information can be obtained during the repairing process, such as the modified programs' validation results. Such information can provide valuable feedback for continuously refining the navigation of the modification points [49]. Therefore, in this work, besides suspiciousness scores, the *fixing scores* of the modification points, which refer to the ability to fix the program by modifying the source code of the corresponding points, are used for navigating modification points in each repair iteration. The fixing scores are continuously measured and updated according to the intermediate validation results of the modified programs.

The intuition is that *if modifying the source code at a modification point mp causes (some of) the initial failed test(s) to be passed, mp could be the correct position of the fault or have relations with the fault.* Otherwise, modifying its source code cannot change the results of the failed tests. The modification point with a high fixing score and high suspiciousness score should be prioritized to attempt in each subsequent repair iteration.

After a modification point is selected, APR tools generate and *select suitable modifications* for that point and evaluate them by executing tests [36, 38, 50]. This dynamic validation is time-consuming and costs a large amount of resources. In order to mitigate the wasted time of validating incorrect modifications, this dissertation introduces *modification suitability measurement* for lightweight evaluating and quickly eliminating unsuitable modifications. The suitability of a modification at position mp is evaluated by the similarity of that modification with the original source code and with the previous attempted modifications at mp. The intuition is that *the correct modification at mp is often similar to its original code and the other successful modifications at this point, while the modifications similar to the failed modifications are often incorrect.* Thus, the more similar a modification is to the original code and to the successful modifications, and the less similar it is to the failed modifications, then the more suitable that modification is for attempting at mp.

These heuristic rules are embedded on both product-based and system-based approaches, and the enhanced versions are called $ProdBased_{enhanced}$ and $SysBased_{enhanced}$.

Several experiments were conducted on a dataset of 318 buggy versions of 5 SPL systems (i.e., 318 variability bugs) to evaluate the performance of the approaches, $ProdBased_{basic}$, $SysBased_{basic}$, $ProdBased_{enhanced}$, and $SysBased_{enhanced}$. The experimental results show that the product-based approach is considerably better than the system-based approach by **12 to 30 times** in the number of plausible fixes and about **20 times** in the number of correct fixes. Interestingly, the heuristics could help to boost the performance of both product-based and system-based approaches by up to **200%**. For instance, by adopting the APR tool Cardumen [63], $ProdBased_{basic}$ and $SysBased_{basic}$ can **correctly fix 13 and 0 systems** respectively, while $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ **correctly fix 40 and 1 systems** respectively. Moreover, the repair performance could be negatively impacted by FL tools since the modification points are selected based on FL results which are often imperfect. To mitigate the impact of the third-party FL tool, this dissertation assesses the effectiveness of the repair approaches if correct FL results

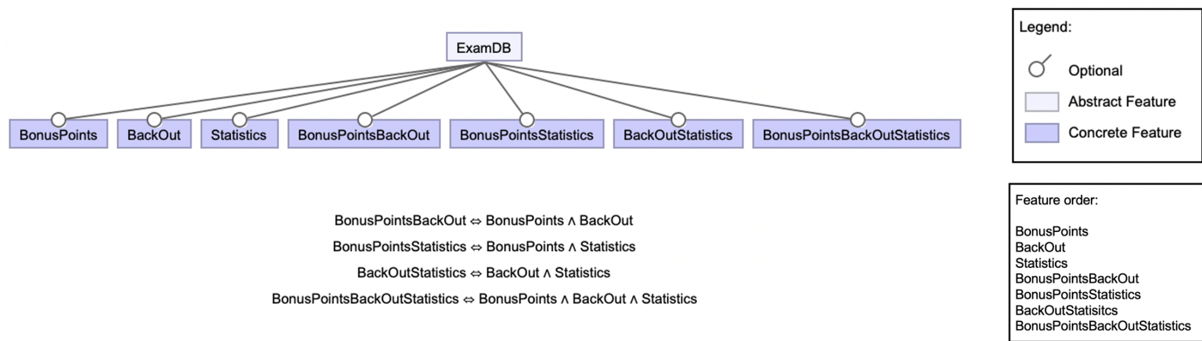


Figure 5.1: The feature model of the ExamDB system

are provided. This experiment shows that the product-based approach is better than the system-based approach about **3 times** in effectiveness and **9 times** in efficiency. In addition, the proposed heuristic rules help to increase **30-150%** the number of correct fixes and decrease **30-70%** the number of attempted modification operations of the corresponding basic approaches.

5.2 Problem Statement

Listing 5.1: Class `ExamDataBaseImpl` in Feature `BackOut`

```
1  public class ExamDataBaseImpl{
2
3  public int getGrade(int matrNr) throws ExamDataBaseException{
4    int i = getIndex(matrNr);
5    if(students[++i] != null && !students[i].backedOut){
6      //Patch:  if(students[i] != null && !students[i].backedOut)
7      return pointsToGrade(students[i].points, 0);
8    }
9    throw new ExamDataBaseException("Matriculation number not found");
10 }
11 }
```

Figure 5.1, Listing 5.1, and Listing 5.2 show a variability bug in the ExamDB system. This system has 8 features, and the bug occurs in the feature named `BackOut` (line 5, Listing 5.1). The *feature model* [7] and *feature order* [158] of this system are defined in Figure 5.1. The feature model defines the dependencies and constraints between features. To construct a concrete product, features are added one after another following the feature order. This SPL system is sampled and tested by 8 products. The corresponding configurations and test results of these products are shown in Table 5.1. This variability

Table 5.1: The tested products of ExamDB system and their test results

Product	ExamDB	BonusPoints	BackOut	Statistics	BonusPointsBackOut	BonusPointsStatistics	BackOutStatistics	BonusPointsBackOutStatistics
p_1	T	F	F	F	F	F	F	F
p_2	T	T	T	F	T	F	F	F
p_3	T	T	F	T	F	T	F	F
p_4	T	F	T	T	F	F	T	F
p_5	T	T	T	T	T	T	T	T
p_6	T	T	F	F	F	F	F	F
p_7	T	F	F	T	F	F	F	F
p_8	T	F	T	F	F	F	F	F

T means that the corresponding feature is enabled and F means that the corresponding feature is disabled in the product.
p_4 and p_8 fail at least one test (*failing products*). Other products pass all their tests (*passing products*).

bug causes products p_4 and p_8 to fail at least one test of their test suites, while the other products pass all their tests, i.e., $P_F = \{p_4, p_8\}$ and $P_P = \{p_1, p_2, p_3, p_5, p_6, p_7\}$.

Listing 5.2: Class `ExamDataBaseImpl` in Feature `BonusPointBackOut`

```
1  public class ExamDataBaseImpl{
2
3  public int getGrade(int matrNr) throws ExamDataBaseException{
4      int i = getIndex(matrNr);
5      if(students[i] != null && !students[i].backedOut){
6        return pointsToGrade( students[i].points, students[i].bonusPoints);
7      }
8      throw new ExamDataBaseException("Matriculation number not found");
9  }
10 }
```

In this system, method `getGrade` of class `ExamDataBaseImpl` is implemented by both features `BackOut` and `BonusPointBackOut`. If `BackOut` is enabled and `BonusPointsBackOut` is disabled, the buggy version of the method `getGrade` is included in the source code of the product and causes the product failure. Instead, if the feature `BonusPointsBackOut` is enabled, the correct version of `getGrade` implemented in this feature (Listing 5.2) will be composed in the product. Note that, even if both `BackOut` and `BonusPointBackOut` are enabled, the correct method `getGrade` of `BonousPointBackOut` will still be included in the product's source code, according to the defined feature order (shown in Figure 5.1). As a result, the products such as p_2 and p_5 have correct behaviors and pass all their tests.

$S = \{ExamDB.ExamDataBaseImpl.58,\ BackOut.ExamDataBaseImpl.26, ...\}$ is the set of suspicious statements detected and ranked by the FL tool VarCop. In particular, $ExamDB.ExamDataBaseImpl.58$ is the statement at line 58 in class `ExamDataBaseImpl` of the feature `ExamDB` (not shown in Listing 5.1). Besides, $BackOut.ExamDataBaseImpl.26$

is statement at line 26 in class `ExamDataBaseImpl` of the feature `BackOut` (i.e., the statement s_5 in Listing 5.1). The repair approach could select these statements as modification points for fixing. The modification of statement $ExamDB.ExamDataBaseImpl.58$ affects all the products of the system since the feature `ExamDB` is enabled in all of the products. Instead, the modification of statement $BackOut.ExamDataBaseImpl.26$ affects the products p_4 and p_8, whose feature `BackOut` is enabled, and they both contain this statement. Although feature `BackOut` is also enabled in two products p_2 and p_5, the statement $BackOut.ExamDataBaseImpl.26$ is not contained by these two products, so modifying it will not affect the behaviors of p_2 and p_5.

Repairing variability bugs in SPL systems means fixing the buggy statements to not only cause all the failing products to pass their tests but also not break the behaviors of any passing products of the systems.

Definition 5.1 ***(Repairing variability faults in an SPL system).*** *Let us consider the 4-tuple* $\langle \mathfrak{S}, P, \mathcal{T}, S \rangle$*, where:*

- $\mathfrak{S}$ *is an SPL system containing variability bugs,*
- $P = \{p_1, \ldots, p_n\}$ *is the set of n sampled products,* $P = P_P \cup P_F$*, where* P_P *and* P_F *are the sets of passing and failing products of* $\mathfrak{S}$*,*
- $\mathcal{T} = \{T_1, \ldots, T_n\}$ *is a set of test suites, where* $T_i \in \mathcal{T}$ *is the test suite of product* p_i *(for each* $i \in \{1, \ldots, n\}$*), and*
- $S = \{s_1, \ldots, s_k\}$ *is the ranked list of suspicious statements of the system* $\mathfrak{S}$ *which could be obtained by an FL technique.*

Repairing variability bugs in an SPL system consists in finding candidate patch(es) which make all the available tests in $\mathcal{T}$ *pass.*

5.3 Automated Variability Fault Repair

This section introduces two approaches to repair variability bugs of an SPL system as defined in Definition 5.1: product-based ($ProdBased_{basic}$) and system-based ($SysBased_{basic}$), which not only try to fix the incorrect behaviors of the failing products but also try to not break the correct behaviors of the passing products.

The *product-based approach* individually repairs each failing product $p_i \in P_F$. Then, the obtained patches, which cause p_i to pass all its tests, are propagated and validated on the other products of the system. By this approach, *during the repair process, only the information of the product under repair* p_i *is considered for generating, evaluating, and/or evolving the patches.* The other products of the system are used to validate the obtained patches after finishing the process of repairing p_i.

For the *system-based approach*, instead of repairing one individual product at a time, all the products in P are considered at the same time during repair. By this method, *in each iteration of the repair process, the patches are generated, evaluated, and/or evolved based on the information of all the products in* P *of the system.*

For both approaches, the valid patches are the patches causing all the available tests of all the products in P to pass.

5.3.1 Product-based Approach ($ProdBased_{basic}$)

The product-based approach ($ProdBased_{basic}$) is shown in Algorithm 5.1. Specifically, an APR tool R is adopted to generate patches for each failing product $p_i \in P_F$ with its contained suspicious statements S_i (line 5-6). Note that S is the list of suspicious statements of the whole system $\mathfrak{S}$, which is identified by an FL technique such as VarCop. For each failing product p_i, each statement $s \in S$ needs to be mapped to the corresponding statements in the product. In other words, S_i (line 5) consists of the suspicious statements which occur in p_i, $S_i \subseteq S$. During the repair of product p_i, only the suspicious statements contained in this product are considered. The process of generating patches `PatchGeneration` (i.e., line 6 in Algorithm 5.1) is described in details latter.

`PatchGeneration` gives as output the set of candidate patches C_i which make all the tests T_i of p_i to pass (line 6). Each candidate patch $c \in C_i$ is then propagated and validated on all the other products in P (line 7). If a candidate patch causes all the tests in $\mathcal{T}$ to pass, then the patch is *valid* for fixing the system $\mathfrak{S}$. If none of the candidate patches in C_i causes all the tests in $\mathcal{T}$ to pass, the process continues by trying to fix the other failing products of the system (line 4).

The fixing process stops if one of the following conditions is satisfied: (i) there is at least a candidate patch passing all the available tests of all products in P, (ii) all the failing products in P_F have been attempted to be fixed by R, or (iii) the time execution limitation

Algorithm 5.1: $ProdBased_{basic}$ algorithm

```
1  Procedure ProductBasedApproach(P, 𝒯, S)
       Input  : P is the set of products of the system 𝔖
                𝒯 is the set of test suites of the products in P
                S is the list of suspicious statements of the system 𝔖
       Output: Set of valid patches (validPatchSet)
2      begin
3          validPatchSet ← ∅
4          for p_i ∈ P_F do
5              S_i ← SuspStmtInProductMapping(S, p_i)
6              C_i ← PatchGeneration(p_i, T_i, S_i)
7              validPatchSet ← PatchGlobalValidation(P, 𝒯, p_i, C_i)
8              if validPatchSet.size() > 0 or timeout then
9                  break
10             end
11         end
12         return validPatchSet
13     end
```

is reached. The complexity of $ProdBased_{basic}$ algorithm (Algorithm 5.1) is $O(n)$ where n is the number of failing products.

Patch generation

Algorithm 5.2 shows the process of generating patches for an individual failing product $p_i \in P_F$ (i.e., line 5 in Algorithm 5.1). In general, any APR tool R can be employed as the `PatchGeneration` procedure. For a selected modification point *mp* (line 6), depending on the employed APR tool, a modification operation can be generated by considering or not some specific information; such information could be taken from the program under repair or from outside the program. Without loss of generality, this algorithm only sets the product under repair p_i as an input for generating modification operations (line 7), and the other information is excluded.

The generated modification operations d are then used to construct candidate patches (line 8). A patch could contain one or multiple modification operations, and it could also be obtained through a cross-over operation or evolved from the previous patches. In this

Algorithm 5.2: Patch generation algorithm in $ProdBased_{basic}$

```
1  Procedure PatchGeneration(p_i, T_i, S_i)
     Input  : p_i is the product under repair
              T_i is the test suite of product p_i
              S_i is the list of suspicious statements in product p_i
     Output: Set of valid patches of product p_i (C_i)
2    begin
3      C_i ← ∅ //valid patches of product p_i
4      testResult ← TestResultInitialization(p_i, T_i)
5      while ¬searchStop do
6        mp ← ModificationPointSelection(S_i)
7        d ← ModificationOperationGeneration(p_i, mp)
8        c ← PatchConstruction(d, testResult)
9        testResult ← PatchLocalValidation(p_i, T_i, c)
10       if ∀t ∈ testResult, t is a passed test then
11         C_i.add(c)
12       end
13     end
14     return Ci
15   end
```

algorithm, `PatchConstruction` is also an abstract function representing the corresponding function of any APR tool. This algorithm sets two inputs for this function, including the modification operation d and the test results *testResult* of the previous patch or of the original product if no patch has been found so far. In practice, several APR approaches such as GenProg [18] need the *testResult* to measure the fitness, navigate the search, and the evolution process during generating patches. Instead, some other APR approaches do not need the *testResult* of the previous patch. For example, jMutRepair [36] can synthesize a new patch in each iteration of the repair process without evolving from the other attempted patches. To keep the algorithm as simple as possible, some other additional information (e.g., the previous attempts) that could be needed in `PatchConstruction` is excluded in this algorithm.

After that, the function `PatchLocalValidation` (line 9) executes the patched product with the given test suite T_i and outputs the corresponding test results. If all the tests in T_i are passed, the corresponding patch is a valid patch of the product p_i (line 10–11). Such

Algorithm 5.3: Patch global validation algorithm in $ProdBased_{basic}$

```
1  Procedure PatchGlobalValidation(P, 𝒯, p_i, C_i)
     Input  : P is the set of products of the system 𝔖
              𝒯 is the set of test suites of the products in P
              p_i is the product under repair
              C_i is the set of valid patches of product p_i
     Output: Set of patches which are valid for all products in P (validPatchSet)
2    begin
3        validPatchSet ← ∅
4        for c ∈ C_i do
5            valid ← true
6            for p_k ∈ P \ {p_i} do
7                testResult_{p_k} ← PatchLocalValidation(p_k, T_k, c)
8                if ∃t ∈ testResult_{p_k}, t is failed test then
9                    valid ← false
10                   break
11               end
12           end
13           if valid then
14               validPatchSet.add(c)
15           end
16       end
17       return validPatchSet
18   end
```

valid patches will be globally validated on the other products of the systems (invocation at line 7 in Algorithm 5.1, and definition in Algorithm 5.3).

Patch validation

The process `PatchGlobalValidation` of propagating and validating the obtained patches of p_i on the other products of the system (line 7 in Algorithm 5.1) is shown in Algorithm 5.3. Different products of an SPL system could share multiple statements. Thus, a fix in one product needs to be propagated and validated by the other products of the system. Specifically, for validating the obtained patch $c \in C_i$ of p_i on the product p_k (being p_k a product in P different from p_i), all of the modifications in c are applied into the

corresponding positions in p_k (line 6–11). Note that there are cases that a modification in c cannot be applied in p_k since there does not exist a corresponding modification point in p_k. If all the modifications in c cannot be applied in p_k, this patch will not impact p_k's behaviors. This means that the validation result of p_k with such a patch corresponds to its original test result. If a patch $c \in C_i$ causes all the tests of all products in P to pass, c is a valid patch to fix the system $\mathfrak{S}$ (lines 13–15).

5.3.2 System-based Approach ($SysBased_{basic}$)

The system-based approach ($SysBased_{basic}$) for repairing a buggy SPL system $\mathfrak{S}$ is shown in Algorithm 5.4. This algorithm attempts to repair all the products of the system $\mathfrak{S}$ at the same time. For each repair iteration, the modification point mp is selected in the ranked list of all the suspicious statements S of the system (line 6). Then, any APR tool R can be used to generate a suitable modification operation for the selected point (line 7). Similarly to Algorithm 5.2, the patch can be constructed by considering only the newly generated modification operation d or evolved from the previous attempts (line 8). After that, the generated patch is applied to all the products $p_i \in P$ of the system and validated by all the test suites $T_i \in \mathcal{T}$ (lines 10–13). For a product p_i which does not contain any corresponding modification points of the modifications in a patch c, $testResult_{p_i}$ (line 11) is exactly the original test results of this product. Indeed, if p_i does not contain the statements modified by c, its source code cannot be changed when c is applied.

To construct a patch, similarly to the product-based approach, the test result of the previous patch is passed to `PatchConstruction` function. However, instead of using the test result of one product, the system-based approach employs the test results of all the products, *testResultSet*, for measuring fitness values and guiding the search process in the next generation (line 8). Any patch which causes all the tests in $\mathcal{T}$ to pass is a valid patch (line 14–23). The repair stops (guard at line 5) when: (i) at least a valid patch is found or (ii) the time budget is exhausted. The complexity of the system-based algorithm (Algorithm 5.4) is $O(m)$ where m is the number of products of the system.

Computational improvement

In each repair iteration, the patches are generated, evaluated, and/or evolved based on the test information of all the sampled products P of the system. Thus, all the products must be validated after each patch c is constructed (line 10–13). However, executing tests

Algorithm 5.4: $SysBased_{basic}$ algorithm

```
1  Procedure SystemBasedApproach(𝔖, P, 𝒯, S)
     Input  : 𝔖 is the SPL system under repair
              P is the set of products of the system 𝔖
              𝒯 is the set of test suites of the products in P
              S is the list of suspicious statement of system 𝔖
     Output: Set of patches which are valid for all products in P (validPatchSet)
2    begin
3      validPatchSet ← ∅
4      testResultSet ← TestResultsInitialization(P, 𝒯)
5      while validPatchSet.size() = 0 and ¬timeout do
6        mp ← ModificationPointSelection(S)
7        d ← ModificationOperationGeneration(𝔖, m_p)
8        c ← PatchConstruction(d, testResultSet)
9        testResultSet ← ∅
10       for p_i ∈ P do
11         testResult_{p_i} ← PatchLocalValidation(p_i, T_i, c)
12         testResultSet.add(testResult_{p_i})
13       end
14       valid ← true
15       for testResult_{p_i} ∈ testResultSet do
16         if ∃t ∈ testResult_{p_i}, t is a failed test then
17           valid ← false
18           break
19         end
20       end
21       if valid then
22         validPatchSet.add(c)
23       end
24     end
25     return validPatchSet
26   end
```

of all products P in each repair iteration is very time-consuming. To boost the efficiency of Algorithm 5.4, one could improve the fitness function and stop validating a patch as soon as there is a patched product that failed. For instance, lines 15–19 could be moved

into the loop in line 10, and a suitable `break` statement added to stop validating other products right after a product fails its test(s).

Although this method could enhance the patch validation efficiency, the precision of the fitness function could be negatively affected. Indeed, the fitness functions used to evaluate the patches are based on the number of passed and failed tests [18, 48, 63]. Early stopping right after a test fails, would cause a loss of information about the non-executed test cases. Therefore, to effectively and efficiently validate patches in $SysBased_{basic}$, it would require a different fitness function that can handle the lack of some test information. This is beyond the scope of this research, and I will investigate it in the future work.

5.3.3 Product-based Approach vs System-based Approach

The main difference between the system-based and product-based approaches is the scope of generating and validating patches in each repair iteration. For each repair iteration *in the system-based approach, the* ***PatchConstruction*** *function synthesizes patches, measures the fitness values, and/or evolves patches by considering the test results of* ***all the products*** P *of the system* (line 8, Algorithm 5.4). Instead, *in the product-based approach, the test results of* ***only the product under repair*** *are considered* (line 8, Algorithm 5.2).

For example, the APR approaches which leverage evolutionary algorithms, such as jGenProg [36], often use a fitness function to guide the evolution of a population of patches throughout a number of generations. Specifically, in a given generation k, the patches with better fitness values will be selected for evolving at generation $k + 1$. The default fitness function in jGenProg counts the number of failed tests. The lower number of failed tests, the better a patch. By adopting this type of APR tool, in the product-based approach, the fitness value is the number of failed tests in T_i of the product under repair p_i only. Instead, in the system-based approach, the fitness value is the total number of failed tests of all the tests in $\mathcal{T}$ of all the products of the system.

For the product-based approach, the employed APR tool R can quickly evaluate the generated patches since the number of tests to execute is much less compared to the system-based approach. However, in the product-based approach, the candidate patches could be biased by the product under repair because they are evaluated using the tests of only one product. Consequently, it could be less effective in generating a good patch for the whole system. In contrast, in the system-based approach, the generated patches could be better in the general context of the system compared since they are evaluated

over all the test suites of all the products. However, the number of tests that must be executed in each iteration could be very large. This could be inefficient in practice.

5.4 Heuristic Rules for Improving the Repair Performance

This section introduces several heuristic rules to improve the APR tools' performance by guiding them to effectively and efficiently navigate modification points and select suitable modifications. Then, this section shows how to apply these heuristic rules on both product-based and system-based approaches to boost their performance in repairing variability bugs.

5.4.1 Heuristic Rules for Improving the Performance of Automated Program Repair Tools

Modification point navigation (*ModNav*)

To decide the positions in the program for synthesizing patches, APR tools often select modification points based on their *suspiciousness scores* measured in advance by FL techniques. In previous studies [125, 159, 160], FL techniques, such as Ochiai [31], have shown their effectiveness in creating the initial ranked list of suspicious statements and narrowing the search space for APR tools. However, the list of suspicious statements could be continuously better (re-)ranked, and the precision of modification point navigation could also be gradually refined by the intermediate results of program repair [49, 161].

Indeed, the validation results of the modification operations can provide valuable feedback for correctly finding the positions of the faulty code elements. The key idea of heuristic rule *ModNav* is that *for a modification point mp, if a modification operation at that point causes the initial failed test(s) to be passed, mp could be the correct position of the fault or have relations with the fault.* Instead, if modifying its code can not change the failure states of the failed tests, probably *mp* is not the correct position of the fault. For example, Table 5.2 shows several modification operations generated to fix the bug in Listing 5.1 and their corresponding validation results. In this table, the numbers of initial failed tests and passed tests are shown in the header of corresponding columns. Each row shows the number of initial failed/passed tests which are still failed/passed after applying the modification operation. Although the modification operation $MO3$ at $BackOut.ExamDataBaseImpl.26$ (i.e., statement s_5 in Listing 5.1) could not make all

Table 5.2: Example of modification operations for fixing the bug at statement s_5 in Listing 5.1

Statement ID	Modification operation	#failed tests (2)	#passed tests (116)
ExamDB.ExamDataBaseImpl.58	MO1	2	90
BackOut.ExamDataBaseImpl.26	MO2	1	115
BackOut.ExamDataBaseImpl.26	MO3	1	116

the tests passed, it helps to decrease the number of failed tests. Initially, there were 2 failed tests, and after applying this modification operation, the number of failed tests decreased to 1. This indicates that the modification operation $MO3$ is still incorrect in fixing the buggy statement, but the selected statement seems to be the correct position to be fixed. It should be attempted to continuously fix in the next iterations.

When applying rule *ModNav*, the navigation is guided by not only the *suspiciousness scores* of the code elements but also by their *fixing scores* measured based on the validation results of the attempted modification operations. The fixing score of a modification point $mp = (pos, c_o)$ indicates the ability to fix the program by modifying source code c_o at the position pos. In each repair iteration, *both suspiciousness and fixing scores are combined to navigate modification points.* In particular, suspiciousness scores are calculated once in advance by off-the-shelf FL tools. Instead, fixing scores are initialized with the same values for all modification points and then continuously updated in each repair iteration after a modification operation is applied and validated.

For a modification point mp, if a modification operation at mp causes all the initial failed tests to pass and does not break any initial passed tests, that modification operation is the valid patch of the program under repair. However, in the case that not all of the tests are passed, the modification operation can still reflect the ability to fix the program at the corresponding point. This study measures the fixing score of a modification point based on the information of to what extent the modification at that point fixes unexpected behaviors (i.e., whether some initial failed tests are fixed (changed to "passed")) and breaks correct behaviours (i.e., whether some initial passed tests are broken (changed to "failed")).

To measure the fixing scores and use them for guiding modification point navigation in the next iterations, the modification operations are divided into three groups based on the

test results of the corresponding modified programs: *SomeFixedNoneBroken*, *SomeFixed-SomeBroken*, and *NoneFixed*. Specifically, *SomeFixedNoneBroken* refers to the modification operations causing some initial failed tests to be passed and do not break any initial passed tests. *SomeFixedSomeBroken* refers to the modification operations causing some initial failed tests to be passed but breaking some initial passed tests. Finally, *NoneFixed* refers to the modification operations which cannot change the state of any initial failed tests. For example, the modification operation $MO1$ in Table 5.2 is categorized in the group *NoneFixed*, since none of the failed tests is fixed. The modification operations $MO2$ and $MO3$ are in group *SomeFixedSomeBroken* and *SomeFixedNoneBroken*, respectively. Both $MO2$ and $MO3$ can fix one initial failed test. However, $MO3$ does not break any initial passed test, while $MO2$ breaks one. This intuitively shows that the ability to fix the program at the corresponding modification point of $MO3$ is better than that of $MO2$.

According to the state of fixing the initial failed tests and breaking the initial passed tests, the order of these groups in terms of ability to fix the program is as follows: (1) *Some-FixedNoneBroken*, (2) *SomeFixedSomeBroken*, and (3) *NoneFixed*. This work follows this order to prioritize the selection of a modification point in the next iterations. This means that the modification point whose modification is in group *SomeFixedNoneBroken* will have the highest priority of being selected (i.e., highest fixing score). Particularly, the fixing score of a modification point is measured as follows:

Definition 5.2 ***(Fixing score)***. *Given a program with test suite $T = T_f \cup T_p$, where T_f is the set of initial failed tests and T_p is the set of initial passed tests, after applying a modification operation $d = op(mp, c_n)$, the fixing_score of the modification point mp is:*

$$fixing_score(mp) = \begin{cases} 2 & \text{if } \exists t \in T_f,\ t \text{ becomes a passed test and } \nexists t' \in T_p,\ t' \text{ becomes a failed test} \\ 1 & \text{if } \exists t \in T_f,\ t \text{ becomes a passed test and } \exists t' \in T_p,\ t' \text{ becomes a failed test} \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

In each repair iteration, the modification point with the highest fixing score will be selected. If the modification points have the same fixing scores, the possibility of being selected is decided based on their suspiciousness scores. In this work, the suspiciousness scores of the modification points are measured by a FL tool, VarCop.

The detailed number of failed and passed tests could also somewhat reflect the fixing ability of the modification points. However, this study does not consider such detailed

information in measuring fixing scores. The reason is that, for a modification operation, the detailed number of passed and failed tests depends on not only the modification point but also on the specific modification (i.e., modified code). Such detailed results could provide wrong indication for evaluating the correctness of the modification points. For example, a modification operation that attempts the correct position but using a wrong modification, could fix only some initial failed tests and break multiple passed tests. Instead, the sign of changing the state of a test from failed to passed is a visible evidence showing that the modification point is a buggy element or related to the bug. Thus, that position should be prioritized to continuously attempt modifying (fixing score is 2 or 1).

Modification suitability measurement (*ModSuit*)

After selecting a modification point, an APR tool generates and selects suitable modification operations to attempt to fix that point. To validate the suitability of a modification operation, all the tests (or, at least, all the failed tests) of the program under repair need to be re-executed. This dynamic validation is repeated multiple times during the repair process. This validation step is often time-consuming and costs a large amount of resources. This section proposes the heuristic rule *ModSuit* that measures the *suitability* of a modification operation without executing tests. This could enhance both the efficiency and the effectiveness of APR tools.

The rule is based on the observation that the previous modifications of a modification point *mp* can provide empirical evidence for APR tools about which modifications should or should not be attempted. The reason is that the previous modifications of *mp* are the modifications that have been attempted and validated by tests. Their validation results can reveal whether similar modifications are suitable for that point or not. Additionally, a correct patch is often similar to the original code [162, 163]. Thus, for the modification point *mp*, its original code and the previous modifications can be leveraged to quickly evaluate the suitability of a newly generated modification. The intuition is that a suitable modification *d* at *mp* could be similar to the original code of *mp* as well as the successful modifications (Definition 5.3) at this point, while it should not be similar to the failed modifications (Definition 5.4) which were tested and shown to be not effective.

Definition 5.3 ***(Successful modification)****. Given a program with the test suite T, a modification operation $d = op(mp, c_n)$ is a* successful modification *if after applying d, each test $t \in T$ is a passing test.*

Definition 5.4 ***(Failed modification)***. *Given a program with the test suite T, a modification operation $d = op(mp, c_n)$ is a* failed modification *if after applying d, there exists a test $t \in T$ that is failing.*

In this work, *the suitability of a new modification operation $d = op(mp, c_n)$ is measured based on the similarity of the modified code c_m (if d is applied) with the original code at mp, as well as the previous successful and failed modifications of this point.* The intuition is that the correct modification at mp is often similar to its original code and the other successful modifications at this point, while the modifications similar to the failed modifications are often incorrect. Therefore, the suitability of a modification operation $d = op(mp, c_n)$ at mp depends on:

(i) the similarity between c_m and the original code c_o,

(ii) the similarity between c_m and the modified code of the previous successful modifications, and

(iii) the diversity between c_m and the modified code of the previous failed modifications.

If the suitability score of d is greater than a threshold θ, this work applies it to the program and validate it by tests. Otherwise, it could be quickly discarded without actual executing tests.

Given a modification point mp and its original source code c_o, let $D = D_f \cup D_s$ be the previous modifications at mp, where D_f is the set of failed modifications, and D_s is the set of successful modifications. For a modification operation $d = op(mp, c_n)$, the *suitability score* of attempting d at mp is measured as Equation 5.2.

$$suitability_score(d) = \frac{\alpha(suit(c_m)) + \beta(1 - unsuit(c_m))}{\alpha + \beta} \tag{5.2}$$

where c_m is the modified code obtained by applying d. In Equation 5.2, $suit(c_m)$ tells how d is suitable to be applied at mp. It depends on the similarity of c_m with the original code, as well with the code obtained by successful modifications. $unsuit(c_m)$, instead, shows how d is unsuitable to be applied at mp, and it is given by the similarity of c_m and the failed modifications. Specifically, $suit(c_m)$ and $unsuit(c_m)$ are measured as Equation 5.3 and Equation 5.4, respectively.

$$suit(c_m) = \max(similarity(c_m, c_o), \max_{d_s \in D_s} similarity(c_m, c_{d_s})) \tag{5.3}$$

$$unsuit(c_m) = \max_{d_f \in D_f} similarity(c_m, c_{d_f}) \tag{5.4}$$

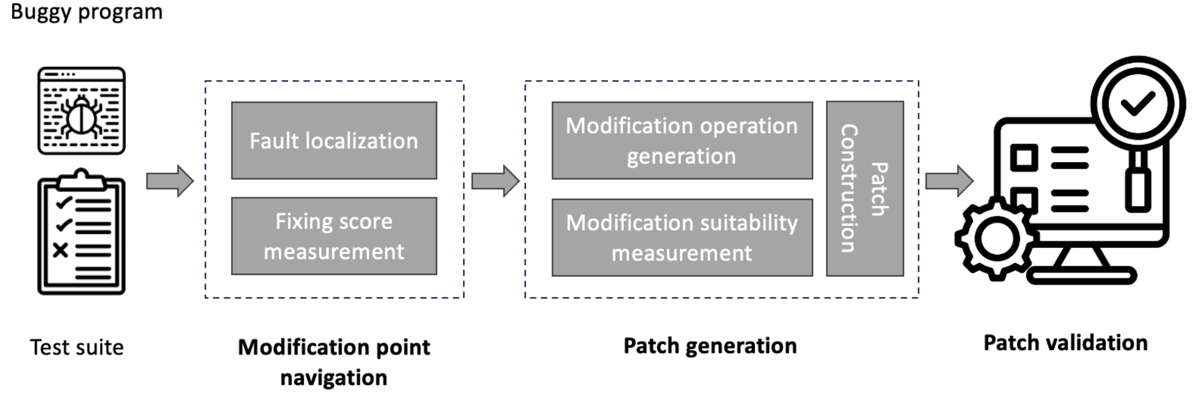


Figure 5.2: The process of APR with the two proposed heuristic rules

In Equation 5.3 and Equation 5.4, c_{d_s} and c_{d_f} are the modified code of a successful operation $d_s \in D_s$ and of a failed operation $d_f \in D_f$, respectively. Overall, if $suitability_score(d, mp) > \theta$, d is applied to the program and validated by executing tests (θ is a hyperparameter of the approach). Otherwise, d is eliminated without actually executing tests. Without loss of generality, any function which can measure the similarity of two sequences can be used in these equations. The impact of different similarity functions is experimented and shown in Section 5.6.2.

5.4.2 Applying the Heuristic Rules in Repairing Variability Faults

Figure 5.2 shows the general process of APR when the proposed heuristic rules are applied. In general, for correctly selecting modification points, both the suspicious scores (measured by FL techniques) and fixing scores (continuously updated based on the intermediate validation results) are leveraged in the modification point navigation step. After that, the APR tool generates modification operations for the selected points. This work does not propose any new modification generation approach, but those proposed by existing APR tools are leveraged; however, this work modifies the tools by adding mechanisms to decide whether to apply the generated modification to the program based on its suitability score. The suitability of a modification is calculated according to its similarity with the original code and the previously attempted modifications of the selected point. If the suitability score exceeds a threshold, this work applies that modification. Otherwise, it is excluded.

Algorithm 5.5: Patch generation algorithm in $ProdBased_{enhanced}$

```
1  Procedure PatchGeneration(p_i, T_i, S_i)
     Input  : p_i is the product under repair
              T_i is the set test suite of product p_i
              S_i is the list of suspicious statements in product p_i
     Output: Set of valid patches of product p_i (C_i)
2    begin
3      C_i ← ∅ //valid patches of product p_i
4      testResult ← TestResultInitialization(p_i, T_i)
5      while ¬searchStop do
6        mp ← ModificationPointSelection(S_i)
7        d ← ModificationOperationGeneration(p_i, mp)
8        score ← ModificationSuitabilityMeasurement(d, mp)
9        if score > θ then
10         c ← PatchConstruction(d, testResult)
11         testResult ← PatchLocalValidation(p_i, T_i, c)
12         if ∀t ∈ testResult, t is a passed test then
13           C_i.add(c)
14         end
15         FixingScoreUpdate(mp, testResult)
16       end
17     end
18     return C_i
19   end
```

$\textbf{ProdBased}_{\textbf{enhanced}}$

For the product-based approach, Algorithm 5.5 shows how the proposed heuristic rules are applied in repairing an individual product of the SPL system, i.e., the procedure of `PatchGeneration` for a product p_i. The algorithm is a modification of the one shown in Algorithm 5.2 and described in Section 5.3.1; the new operations are highlighted in yellow, while the other functions are the same. In this variant of `PatchGeneration`, after each repair iteration, the fixing scores of the modification points are updated (line 15) and then they are combined with the suspiciousness scores for precisely selecting modification points in the next iterations (line 6). In addition, the suitability score of

each modification operation is measured (line 8) and checked with a threshold (line 9) before actually applying it and validating it by executing tests (line 10–14).

Furthermore, in $ProdBased_{enhanced}$, instead of gradually selecting failing products (in a random manner as $ProdBased_{basic}$) for repairing, this work sorts and selects the failing products of the system based on their complexity (***Failing product navigation (ProdNav)***). This dissertation hypothesizes that the less complex a product is, the easier the product can be fixed. This work measures the complexity of a failing product $p_i \in P_F$ based on its source code size and its tests. The intuition is that the bigger the product is, the more complex it is. Also, the more failed tests the product has, the more incorrect behaviors it has. Particularly, the complexity of a product $p_i \in P_F$ having test suite $T_i = T_f \cup T_p$ is measured by Equation 5.5. In this equation, $productSize_{p_i}$ and $systemSize$ are measured at the statement level. $systemSize$ is the total number of all the statements in all the features of system $\mathfrak{S}$. $productSize_{p_i}$ is the number of statements contained in product p_i.

$$complexity(p_i) = \frac{\frac{productSize_{p_i}}{systemSize} + \frac{|T_f|}{|T_i|}}{2} \tag{5.5}$$

SysBased$_{\textbf{enhanced}}$

Algorithm 5.6 shows the details of how the heuristics are applied in the system-based approach. The algorithm is a modification of $SysBased_{basic}$ shown in Algorithm 5.4, and described in Section 5.3.2; the new operations are highlighted in yellow, while the other functions are the same. The main difference of this variant compared to Algorithm 5.4 is the addition of the fixing scores and modification suitability measurement. After each repair iteration, the fixing scores of the modification points are updated (line 23), and then they are combined with the suspiciousness scores for precisely selecting modification points in the next iterations (line 6). In $SysBased_{enhanced}$, the fixing scores are measured based on the results of all the tests in $\mathcal{T}$ of all the products. Additionally, the suitability score of each modification operation is measured (line 8) and checked with a threshold (line 9) before actually applying and validating by executing tests.

5.5 Experiment Methodology

To evaluate the proposed approaches in repairing variability bugs of SPL systems, this chapter aims to answer the following research questions:

Algorithm 5.6: *$SysBased_{enhanced}$* algorithm

```
1  Procedure SystemBasedAproach(𝔖, P, 𝒯, S)
     Input  : 𝔖 is the SPL system under repair
              P is the set of products of the system 𝔖
              𝒯 is the set of test suites of the products in P
              S is the list of suspicious statement of system 𝔖
     Output: Set of patches which are valid for all products in P (validPatchSet)
2    begin
3      validPatchSet ← ∅
4      testResultSet ← TestResultsInitialization(P, 𝒯)
5      while validPatchSet.size() = 0 and ¬timeout do
6        mp ← ModificationPointSelection(S)
7        d ← ModificationOperationGeneration(𝔖, m_p)
8        score ← ModificationSuitabilityMeasurement(d, mp)
9        if score > θ then
10         c ← PatchConstruction(d, testResultSet)
11         testResultSet ← ∅
12         for p_i ∈ P do
13           testResult_{p_i} ← PatchLocalValidation(p_i, T_i, c)
14           testResultSet.add(testResult_{p_i})
15         end
16         valid ← true
17         for testResult_{p_i} ∈ testResultSet do
18           if ∃t ∈ testResult_{p_i}, t is a failed test then
19             valid ← false
20             break
21           end
22         end
23         FixingScoreUpdate(mp, testResultSet)
24         if valid then
25           validPatchSet.add(c)
26         end
27       end
28     end
29     return validPatchSet
30   end
```

Table 5.3: Benchmarks

SPL System	#Products	#Avg. tests/product	#Buggy versions of the SPL
ZipMe	25	255	55
GPL	99	87	105
ExamDB	8	166	49
Email	27	86	36
BankAccount	34	20	73

- **RQ1: Performance Analysis:** How effective and efficient are the proposed approaches (i.e., $ProdBased_{basic}$, $SysBased_{basic}$, $ProdBased_{enhanced}$, $SysBased_{enhanced}$) in repairing buggy SPL systems?
- **RQ2: Intrinsic Analysis:** How do the heuristic rules including *failing product navigation* (*ProdNav*), *modification point navigation* (*ModNav*), *modification suitability measurement* (*ModSuit*) contribute to the repair performance? How do the hyper-parameters α and β of modification suitability measurement (*ModSuit*) impact the repair performance?
- **RQ3: Sensitivity Analysis:** How do the characteristics of the SPL systems, such as the different systems, the number of failing products, and the number of suspicious statements affect the repair performance?

5.5.1 Benchmarks

To evaluate the proposed variability bug repair approaches, several experiments were conducted on a public set of benchmarks of variability bugs [5]. Table 5.3 shows the detailed information the dataset which is used in the experiments of this research. In total, there are 318 buggy SPL systems, and each of them is considered as input of the proposed repair approaches.

5.5.2 Evaluation Procedure and Metrics

Evaluation procedure

Performance analysis.

Experiment setting: The experiments analyzed the performance of repairing variability bugs of all four proposed approaches: $ProdBased_{basic}$, $SysBased_{basic}$, $ProdBased_{enhanced}$, and $SysBased_{enhanced}$ in two settings:

- *with fault localization* (*withFL*): in this setting, the FL approach VarCop was employed to detect and rank suspicious statements. Then, the output of VarCop is used as the input for all four repair approaches.
- *without fault localization* (*withoutFL*). This setting aims to analyze the performance of the APR approaches without the possible negative impact of the third-party FL tool; therefore, this experiment provides as input to the repair approaches only the statements which are actually buggy.

In both settings *withFL* and *withoutFL*, the time execution limitation of all the approaches (i.e., *timeout* in Algorithm 5.1, Algorithm 5.4, and Algorithm 5.6) for fixing a buggy SPL system is 60 minutes. The *searchStop* condition of `PatchGeneration` (Algorithm 5.2 and Algorithm 5.5) is also set to 60 minutes.[1]

APR tool selection: Two representative APR tools R, jGenProg [36] and Cardumen [63] were selected to used as underlying repair tools of the proposed approaches, as explained in Section 5.3. The two APR tools generate modifications at different levels of code elements in Java programs. Specifically, jGenProg is the implementation of the popular APR tool GenProg [18] for Java programs. This tool repairs buggy programs at the *statement-level*. It synthesizes patches by taking code statements from the program under repair and uses an evolutionary strategy to navigate the search space. This means that, for each generation, a number of *best* patches (assessed by a fitness function) are selected for evolution in the next generation. Cardumen, instead, aims at fixing fine-grained code elements, i.e., it operates at the *expression-level*. This tool mines the source code of the program under repair to create repair templates and then uses such templates for synthesizing patches. In order to generate concrete patches, the placeholders in the templates are replaced by variables which frequently occur in the modification points. In

[1]Note that the comparison between product-based and system-based approaches is fair, as both can use at most 60 minutes.

Cardumen, a selective search strategy is used to explore the search space. Particularly, to speed up the search, Cardumen utilizes a probability model to prioritize patches.

There are several reasons for choosing jGenProg and Cardumen. First, they are popular and representative APR tools for Java programs which are widely used in the related studies [61, 161, 164–166]. Second, they target fixing buggy code at different levels, statement and expression levels. Third, they leverage different search strategies, evolutionary and selective. Finally, they are standalone tools which can be executed if provided with the Java program source code and a test suite, without requiring any additional data/components (e.g., ssFix [126] needs to connect to a private engine or LSRepair [167] requires run-time code search over Github repositories).

Statistical analysis: The performance of the approaches could be affected by the randomness of APR tools, e.g., the random selection of modification operators and of the templates, etc. To mitigate the impact of such randomness, each experiment is executed five times. The statistical analysis is conducted in the experimental results of the setting *withFL* whose search space is large and could be highly impacted by the randomness. Note that the FL phases of these experiments are done only once by VarCop, as its result is deterministic.

This chapter compares whether the distributions of five results of #Correct fixes (Section 5.5.2) of the different executions of the enhanced and basic versions of the approaches ($ProdBased_{enhanced}$ vs $ProdBased_{basic}$ and $SysBased_{enhanced}$ vs $SysBased_{basic}$) are significantly different by the Mann-Whitney U test [168]. The confidence value is $\alpha = 0.05$. The distributions are considered to be significantly different if the $p - value$ returned by the Mann-Whitney U test is lower than this confidence value. Otherwise, there is no significant difference. In case of significant difference, the Vargha and Delaney's $\hat{A}_{12}$ effect size is used to assess the strength of the significance. Specifically, if $\hat{A}_{12}$ of approach A and B is greater than 0.5, the results of A are significantly higher (and so better) than those of the other.

Intrinsic analysis. This experiment studied the impacts of the heuristic rules: *Failing product navigation* (*ProdNav*), *Modification point navigation* (*ModNav*), *Modification suitability measurement* (*ModSuit*) by alternatively disabling one of them in $ProdBased_{enhanced}$. In addition, this experiment created different variants of $ProdBased_{enhanced}$ with different similarity functions, suitability parameters, and suitability thresholds (Equation 5.2, Section 5.4.1) in measuring the suitability of the modification operations. These experi-

ments were conducted on the different variants of $ProdBased_{enhanced}$ since this approach obtained the best performance compared to the other approaches (Section 5.6.1). Moreover, due to the time limitation and the need to repeat each experiment multiple times, these experiments were conducted on the variability bugs of three systems, BankAccount, Email, and ExamDB.

Sensitivity analysis. This experiment studied the results of $ProdBased_{enhanced}$ in repairing variability bugs in different SPL systems. This experiment also studied the impact of the following factors: *the number of failing products* and *the number of suspicious statements* of each buggy SPL system. This experiment categorizes the buggy systems into different groups based on their *number of failing products* (resp. *number of suspicious statements*) and analyze the repair performance of $ProdBased_{enhanced}$ of each group.

Metrics

For evaluating the effectiveness of the approaches, *#Plausible fixes* and *#Correct fixes* [38, 164, 169, 170] were adopt. Specifically, *#Plausible fixes* represents the number of buggy systems obtaining valid patches that make the systems pass all their tests. Instead, *#Correct fixes* indicates the number of buggy systems obtaining valid patches that are also equivalent to the ground-truth patches provided by the benchmark. The number of plausible fixes was widely used to show how effective an APR technique is: the higher the number of plausible fixes, the better the APR tool [18, 64]. However, a plausible patch could still be incorrect due to the inadequacy of the provided test suites [65–68]. Thus, in recent studies [38, 170], the number of correct fixes has become a more popular metric: the higher the number of correct fixes, the better the APR approach.

In addition, *#Mods/system* and *Runtime/system* are used for evaluating the efficiency of the approaches. *#Mods/system* shows the average number of modification operations that a repair approach attempts to fix a buggy SPL system. *Runtime/system* reports the average running time taken by a repair approach to fix a buggy SPL system. Indeed, the running time has been widely used to compare the efficiency of the APR tools [159, 160, 171]. However, this metric could be unstable and dependent on many variables that are unrelated to the APR approaches [61]. Thus, this work reports both the average number of attempts and the average runtime that an APR approach takes to repair a buggy SPL system to show how efficient the approach is. The less number of modifications and the less runtime, the better APR approach.

Table 5.4: RQ1 – The performance of repairing variability bugs of the approaches in the setting *withoutFL* (i.e., the correct positions of buggy statements are given)

APR Tool	Metrics	Product-based approach		System-based approach	
		$ProdBased_{basic}$	$ProdBased_{enhanced}$	$SysBased_{basic}$	$SysBased_{enhanced}$
jGenProg	#Correct fixes	38	40	12	28
	#Plausible fixes	64	49	37	34
	#Mods/system	95	27	18	14
	Runtime/system (min)	8.7	6.7	42.1	22
Cardumen	#Correct fixes	40	65	9	24
	#Plausible fixes	112	92	42	41
	#Mods/system	73	20	13	10
	Runtime/system (min)	5.3	3.9	75.1	52

5.6 Experimental Results

5.6.1 RQ1. Performance Analysis

Setting *withoutFL*

The performance of the FL technique has a major impact on the effectiveness and efficiency of the APR approaches. The reason is that the APR tools often select the modification points based on the output of the FL technique (i.e., the suspicious statement ranked lists) and this could contain statements that are not actually faulty. In this experiment, to mitigate the impact of the FL phase, this study conducted an experiment in which, for each buggy SPL system, the APR approaches are given the correct position of the faulty statements in advance. Table 5.4 shows the performance of the APR approaches in this setting on the total of 318 variability bugs. Overall, *the performance of the product-based approaches is better than that of the system-based approaches, about 3 times in effectiveness and 9 times in efficiency*. In addition, *the heuristic rules help to enhance the corresponding basic approaches from 30 to 150% in the number of correct fixes and from 30 to 70% in the number of modification operations*.

For instance, by leveraging jGenProg to generate patches, $ProdBased_{basic}$ can correctly fix 38 buggy systems. Instead, only 12 buggy systems can be correctly fixed by $SysBased_{basic}$. Using Cardumen, $SysBased_{basic}$ can correctly fix 9 buggy systems, however, $ProdBased_{basic}$ can do that for a significantly larger number of systems, i.e., 40 systems. Moreover, the

product-based approach is also much more efficient than the system-based approach. Particularly, the *Runtime/system* of $ProdBased_{basic}$ is 5-14 times lower than $SysBased_{basic}$. These results show that $ProdBased_{basic}$ can correctly fix many more variability bugs in significantly less running time than $SysBased_{basic}$.

One of the main reasons why the product-based approach obtained better results than the system-based approach is that it can attempt more modification operations, which helps to increase its possibility of reaching plausible/correct patches. On average, $ProdBased_{basic}$ tried about 73–95 modification operations on a buggy system. Instead, $SysBased_{basic}$ tried 13-18 modification operations, around five times less than $ProdBased_{basic}$. Indeed, for each modification operation, the product-based approach can quickly evaluate it by the test suite of one product. In contrast, the system-based approach needs to validate the modification over all the test suites of all the sampled products. This dynamic validation process is time-consuming. As a result, given the same limitation of execution time, $ProdBased_{basic}$ can attempt many more modification operations and obtain more plausible/correct fixes.

Although attempting a larger number of modifications, the product-based approach is still more efficient in terms of running time. In particular, to repair an SPL system, $ProdBased_{basic}$ takes about 7 minutes, while $SysBased_{basic}$ takes about 59 minutes. This demonstrates that executing all the test suites of the system to validate the modifications in each repair iteration is really ineffective. To improve the performance of the system-based approach, modifications could be validated by a subset of selected test cases.

Furthermore, the performance of $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ is considerably better than the corresponding basic approaches in both effectiveness and efficiency. For example, with Cardumen, $ProdBased_{enhanced}$ needs to attempt 20 modification operations for each buggy SPL system to correctly fix 65 bugs. Instead, $ProdBased_{basic}$ must attempt 73 modification operations in each system and correctly fixes only 40 bugs. Besides, $SysBased_{enhanced}$ can correctly fix 24 bugs by attempting 10 modifications per system. In contrast, with a larger number of modifications (i.e., 13 modifications per system), $SysBased_{basic}$ can correctly fix a lower number of bugs, only 9 bugs.

In this setting, the suspicious statement lists of the SPL systems contains only buggy statements. Thus, the modification points are always correctly selected by all the approaches in *modification point navigation* step. However, for the same selected modification points, the *ModSuit* heuristics helps $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ to

Table 5.5: RQ1 – The performance of repairing variability bugs of the approaches in the setting *withFL*

APR Tool	Metrics	Product-based approach		System-based approach	
		$ProdBased_{basic}$	$ProdBased_{enhanced}$	$SysBased_{basic}$	$SysBased_{enhanced}$
jGenProg	#Correct fixes	20	37	1	5
	#Plausible fixes	49	49	4	7
	#Mods/system	144	179	20	30
	Runtime/system (min)	19.3	16.2	41.8	44.3
Cardumen	#Correct fixes	13	40	0	1
	#Plausible fixes	92	80	3	4
	#Mods/system	133	122	23	22
	Runtime/system (min)	13.8	15.3	41.3	52.2

avoid attempting a lot of incorrect modification operations and then quickly find the correct ones. Instead, various incorrect modifications are attempted by $ProdBased_{basic}$ and $SysBased_{basic}$. Consequently, $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ can plausibly/-correctly fix more bugs than $ProdBased_{basic}$ and $SysBased_{basic}$. In addition, by avoiding attempting incorrect modifications, $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ can also reduce the number of modifications and consume less running time to fix a buggy SPL system compared to the corresponding basic approaches.

Setting *withFL*

Table 5.5 shows the repairability performance of the proposed APR approaches on the total of 318 variability bugs in the setting *withFL*. In this setting, *the product-based approaches also obtain considerably better results than the system-based approaches.* They obtain significantly higher values for *#Correct fixes* with a significantly lower *Runtime/system*. Furthermore, *the heuristic rules of ModNav and ModSuit can help to significantly boost the effectiveness of both product-based and system-based directions up to five times.*

By both APR tools jGenProg and Cardumen, the effectiveness of $ProdBased_{basic}$ is higher than that of $SysBased_{basic}$ from 12 to 30 times. For instance, by leveraging jGenProg to generate patches, $ProdBased_{basic}$ can plausibly fix 49 systems. Instead, only 4 systems can be plausibly fixed by $SysBased_{basic}$. By using Cardumen, $SysBased_{basic}$ cannot correctly fix any buggy system, while $ProdBased_{basic}$ can do that for 13 systems. In

addition, the product-based approach is also much more efficient than the system-based approach. The *Runtime/system* of $ProdBased_{basic}$ is 2-3 times lower than $SysBased_{basic}$. Similarly to the discussion for the setting *withoutFL*, the product-based approach locally validates each modification by the tests of one product before globally validating it by the whole test suites of the system. Thus, it can attempt more modification operations in less running time and efficiently obtain a higher number of plausible/correct fixes.

Notably, *by applying the heuristic rules,* $ProdBased_{enhanced}$ *improves the number of correct fixes of* $ProdBased_{basic}$ *from 85% to 207%, and* $SysBased_{enhanced}$ *improves the performance of* $SysBased_{basic}$ *up to 5 times.* For example, using Cardumen, 40 systems can be correctly fixed by $ProdBased_{enhanced}$, and $SysBased_{enhanced}$ can correctly fix 1 system; instead, the corresponding numbers of correct fixes of $ProdBased_{basic}$ and $SysBased_{basic}$ are 13 and 0, respectively. Indeed, the validation results of the previous modification operations are very valuable in guiding the APR approaches to correctly select modification points and avoid attempting unsuitable modifications. For example, with the bug in Listing 5.1, after attempting modification $MO3$ (Table 5.2), the associated modification point of the buggy statement $mp = (BackOut.ExamDataBaseImpl.26, if(students[++i]! = null\&\&!student[i].backedOut))$ is prioritized by $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ to be continuously fixed. Also, the previous attempted modification operations at this point, yet incorrect such as $d = ins_aft(mp, this.students = new\, main.Student[100])$ or the other modification $d = ins_bef(mp, main.Student[]\, oldStudents = students)$ provide feedback to help $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ avoid attempting similar incorrect modifications in the next iterations. As a result, the correct modification operation for this buggy statement is sooner attempted by $ProdBased_{enhanced}$ and $SysBased_{enhanced}$.

Moreover, *the efficiency of the enhanced approaches and the corresponding basic approaches is not significantly different, and it greatly depends on the employed APR tools.* For instance, using jGenProg, $ProdBased_{enhanced}$ takes less runtime to attempt more modifications to fix a buggy SPL system than $ProdBased_{basic}$. For instance, for a buggy SPL system, $ProdBased_{enhanced}$ attempts 179 modifications in 16 minutes, while $ProdBased_{basic}$ takes 19 minutes to attempt 144 modifications. In contrast, leveraging Cardumen, compared to $ProdBased_{basic}$, $ProdBased_{enhanced}$ takes a slightly higher runtime to attempt less number of modifications. $ProdBased_{basic}$ spends about 14 minutes on 133 modifications to fix a variability bug, while $ProdBased_{enhanced}$ spends about 15

Table 5.6: RQ1 – Statistical analysis regarding #Correct fixes of $ProdBased_{enhanced}$ vs $ProdBased_{basic}$ and $SysBased_{enhanced}$ vs $SysBased_{basic}$ in different experiment executions – *withFL* setting

	Product-based approach		System-based approach	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
jGenProg	0.01	1	0.02	1
Cardumen	0.01	1	0.08	-

minutes to attempt 122 modifications.

In fact, the number of modification operations is not always proportional to the runtime. The reason is that for a system, the modification operations of the approaches could try to modify code of different modification points. In setting *withFL*, the modification points are selected on the list of suspicious statements returned by VarCop, which could contain not only buggy but also non-buggy statements. With the guidance of the heuristics *ModNav*, the modification points selected by the enhanced approaches are different from the corresponding basic ones. The differences in the selected modification points could lead to the number of affected products being different, which leads to the variation in the runtime of the approaches; this is because only the products affected by the modifications need to re-execute the tests. However, by not being significantly different in *#Mods/system* and *Runtime/system*, $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ can correctly fix a much higher number of buggy systems compared to the basic approaches.

Statistical analysis

In general, *the enhanced variants of both product-based and system-based approaches consistently obtain significantly better effectiveness than the corresponding basic variants.* Table 5.6 shows the statistical analysis of the differences of $ProdBased_{enhanced}$ compared to $ProdBased_{basic}$, and $SysBased_{enhanced}$ compared to $SysBased_{basic}$ regarding the number of correct fixes. For product-based approaches, the number of correct fixes of $ProdBased_{enhanced}$ and $ProdBased_{basic}$ is significantly different ($p - value = 0.01$). Also, in all the experiment executions, $ProdBased_{enhanced}$ always obtains higher results ($\hat{A}_{12} = 1$). Similarly, $SysBased_{enhanced}$'s performance is considerably better than $SysBased_{basic}$ if jGenProg is used. However, by using Cardumen, the results of these

Table 5.7: RQ2 – Impact of disabling each heuristic rule in $ProdBased_{enhanced}$

	$ProdBased_{enhanced}$	Not applied rule		
		ProdNav	*ModNav*	*ModSuit*
#Correct fixes	34	32	32	14
#Plausible fixes	54	51	48	63
#Mods/system	148	150	147	158
Runtime/system (min)	2.2	1.8	1.8	2.3

system-based approaches are not significantly different. The reason is that the results of Cardumen in these experiments are not good, only 1 or none of the systems can be fixed. Thus, it cannot show the differences in the approaches' results.

5.6.2 RQ2. Intrinsic Analysis

Impact of the heuristic rules

Table 5.7 shows how the heuristic rules impact the performance of $ProdBased_{enhanced}$. As expected, $ProdBased_{enhanced}$ *obtains the highest performance when all the heuristic rules are enabled*. Its effectiveness could decrease from 6% to 60% if one of the heuristics is disabled. Particularly, $ProdBased_{enhanced}$ can correctly fix 34 bugs in three experimental systems, BankAccount, Email, and ExamDB. If *ProdNav* or *ModNav* are disabled, the number of correct fixes is 32. Instead, if *ModSuit* is disabled, only 14 variability bugs can be correctly fixed.

Indeed, if the *ModSuit* is disabled, more modifications are attempted and so the number of plausible fixes can significantly increase. For instance, $ProdBased_{enhanced}$ obtains 54 plausible fixes, while this number is increased by 17% (63 plausible fixes) if *ModSuit* is disabled. This result indicates that without lightweight evaluating and eliminating incorrect modifications, there is a high rate of over-fitting or false-positive patches. Indeed, in this variant of $ProdBased_{enhanced}$, multiple incorrect modification operations have been attempted. These modification operations can cause the system to pass all the available tests, but they are incorrect patches.

Moreover, the running time of $ProdBased_{enhanced}$ does not significantly vary when dis-

Table 5.8: RQ2 – Impact of the similarity functions in modification suitability measurement

Metrics	Cosine	N-gram	Longest common subsequence	Levenshtein	Jaccard
#Correct fixes	28	31	32	34	36
#Plausible fixes	52	47	58	54	56
#Mods/system	154	154	153	148	145
Runtime/system (min)	2.2	2.3	2.0	2.1	2.3

abling the heuristic rules. For instance, $ProdBased_{enhanced}$ takes about 2.2 minutes to repair a buggy SPL system. If one of the heuristic rules is disabled, the fixing process of this approach consumes from 1.8 to 2.3 minutes. Indeed, enabling and disabling the heuristic rules affect the selected modification points in the repair process. This leads to the numbers of affected products of each system being different. Thus, the runtime slightly fluctuates in these experiments due to the differences in the number of the affected and tested products.

Impact of the similarity functions in *ModSuit*

This experiment built different variants of $ProdBased_{enhanced}$ with different similarity functions for measuring the suitability of the modification operations in *ModSuit* (Equation 5.3 and 5.4, Section 5.4.1). As seen in Table 5.8, $ProdBased_{enhanced}$ *obtains the highest number of correct fixes when the Jaccard is used as the similarity function.* Instead, *when Cosine or N-gram is leveraged,* $ProdBased_{enhanced}$*'s effectiveness is lowest, although the number of modification operations is the highest.* For instance, by Jaccard, the number of correct fixes is 36, while the results of Cosine and N-gram are 28 and 31, respectively.

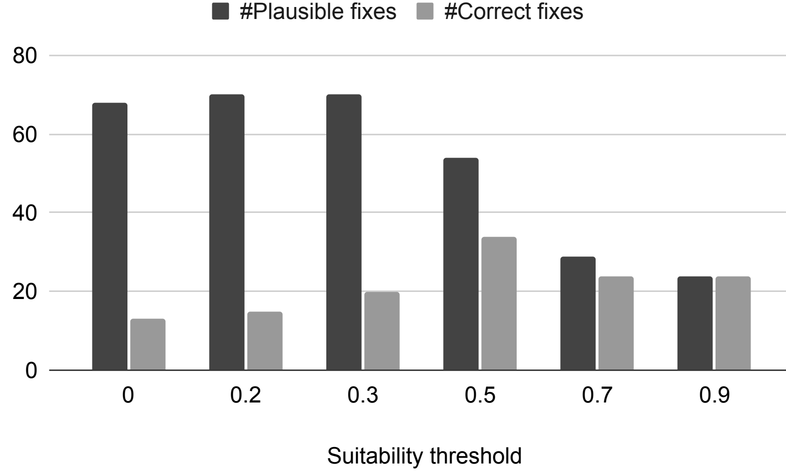
There are two reasons for these results of the different similarity functions. First, multiple bugs can be fixed by minor changes [163]. That is the reason why similarity functions like Levnshtein or Jaccard, which reflect the edit distances of the sentences, could be helpful. Second, the order of the tokens in the modified code of the modifications could not be an

important factor in deciding the similarity in the program repair topic. For example, the correct modification and the buggy source code could share several variables, operators, etc., but these tokens are in different orders in the code statements. Metrics such as Cosine or N-gram, which consider the order of tokens in measuring the similarity, could not work well in this case.

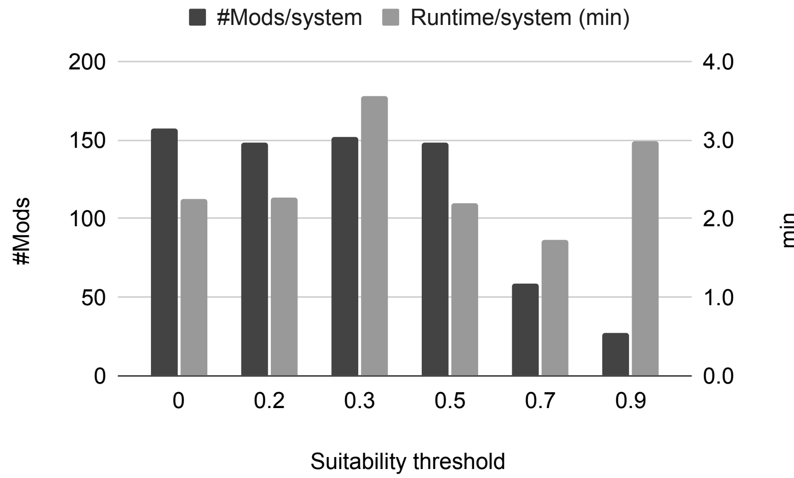
Impact of the suitability threshold θ in *ModSuit*

Figure 5.3 shows the repair performance of $ProdBased_{enhanced}$ with different values of threshold θ in deciding whether a modification is suitable for applying into the program (rule *ModSuit*). In general, *the number of plausible fixes decreases when the threshold increases* (see Figure 5.3a). Specifically, if θ is equal to 0, the number of plausible fixes is 68, while if θ is 0.9, this number is 24. This is because the higher the threshold is, the more strict *ModSuit* is in evaluating the suitability of the modifications. This leads to multiple modifications being eliminated without being applied to the system and validated against the tests. Indeed, as shown in Figure 5.3b, increasing the threshold leads to a considerable decrease in the number of modification operations, from 157 modifications when θ is 0, to 27 modifications when θ is 0.9. Therefore, the number of plausible fixes decreased.

The previous observations can be explained as follows. *$ProdBased_{enhanced}$ obtains the highest number of correct fixes with $\theta = 0.5$* (see Figure 5.3a). For instance, if the threshold increases from 0 to 0.5, the number of correct fixes increases from 13 to 34. However, if the threshold continuously increases from 0.5 to 0.9, the number of correct fixes decreases from 34 to 24. Indeed, with the small value of the threshold $\theta = 0$, multiple incorrect modification operations could be attempted, and then APR tools stop searching when a plausible fix of the system is reached. However, that fix could be still incorrect. Thus, although the number of plausible fixes is high, multiple of those fixes are over-fitting and incorrect. On the other hand, with the large threshold value $\theta = 0.9$, the approach could eliminate multiple promising modification operations. This leads to the decrease in the numbers of both plausible and correct fixes. *Overall, to ensure that the best performance of $ProdBased_{enhanced}$, the threshold θ should be* 0.5.



(a) Effectiveness (#Plausible fixes, #Correct fixes)

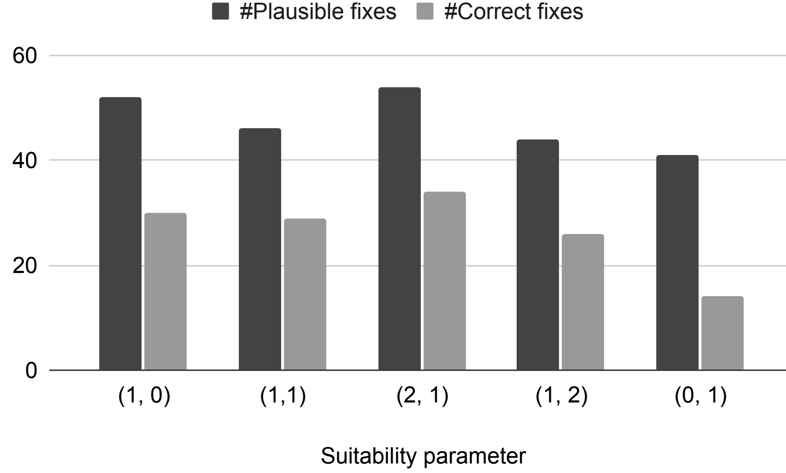


(b) Efficiency (#Mods/system, Runtime/system)

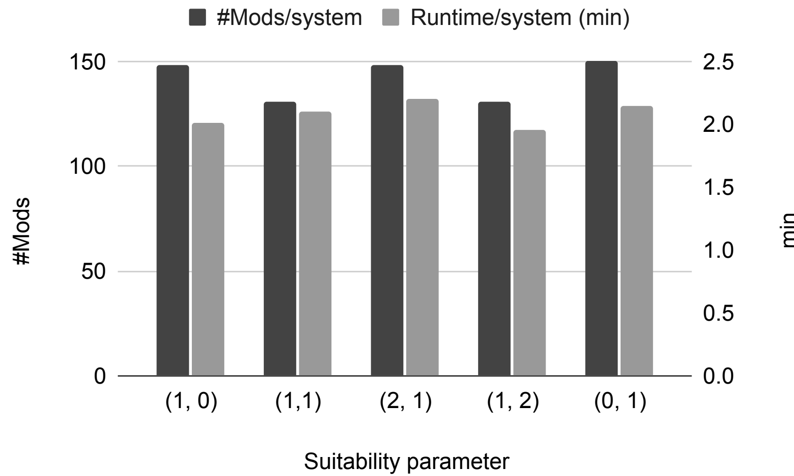
Figure 5.3: RQ2 – Impact of the suitability threshold θ on $ProdBased_{enhanced}$'s performance

Impact of the hyperparameters α and β in *ModSuit*

In order to analyze how the suitability parameters impact $ProdBased_{enhanced}$'s performance, experiments were conducted with five different values of the pair (α, β) in Equation 5.2, as shown in Figure 5.4. Specifically, for a modified code c_m, if the pair (α, β) is $(1, 0)$ or $(0, 1)$, it means that the value of only $suit(c_m)$ or $unsuit(c_m)$ is used to evaluate its suitability. Instead, when both α and β are greater than 0, it means that both $suit(c_m)$ and $unsuit(c_m)$ are considered in the evaluation, with different importance; using $(2, 1)$ gives more importance to $suit(c_m)$, while using $(1, 2)$ gives more importance to



(a) Effectiveness (#Plausible fixes, #Correct fixes)



(b) Efficiency (#Mods/system, Runtime/system)

Figure 5.4: RQ2 – Impact of the suitability parameters (α, β) on $ProdBased_{enhanced}$'s performance

$unsuit(c_m)$. Finally, when the setting is $(1, 1)$, it means that the two aspects have the same importance.

These different parameters do not significantly affect $ProdBased_{enhanced}$'s efficiency, as the numbers of modification operations and running time just vary slightly, as demonstrated in Figure 5.4b. However, $ProdBased_{enhanced}$'s *effectiveness is considerably affected*. $ProdBased_{enhanced}$ obtains the highest *#Correct fixes* when both $suit(c_m)$ and $unsuit(c_m)$ are considered and $suit(c_m)$ has a higher priority ($\alpha = 2$ and $\beta = 1$) (Figure 5.4a). In addition, the repairing result of $ProdBased_{enhanced}$ is the lowest when $\alpha = 0$

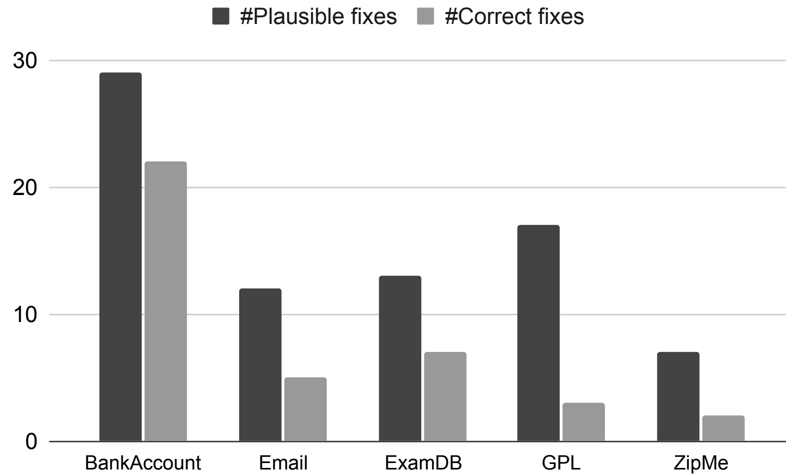
and $\beta = 1$. This variant of $ProdBased_{enhanced}$ can correctly fix 14 buggy systems, half of those fixed when both $suit(c_m)$ and $unsuit(c_m)$ are considered. Indeed, when $\alpha = 0$, the suitability of a new modification is evaluated by only considering the feedback of the previous failed attempts without considering the original code of the corresponding modification point. The previous attempts could guide $ProdBased_{enhanced}$ to decide incorrect modifications, i.e., the modifications similar to the failed modifications can be eliminated. However, by only previous failed attempts, there is little or even no evidence about correct modifications, which is reflected by the similarity with the original code [162, 163].

5.6.3 RQ3. Sensitivity Analysis

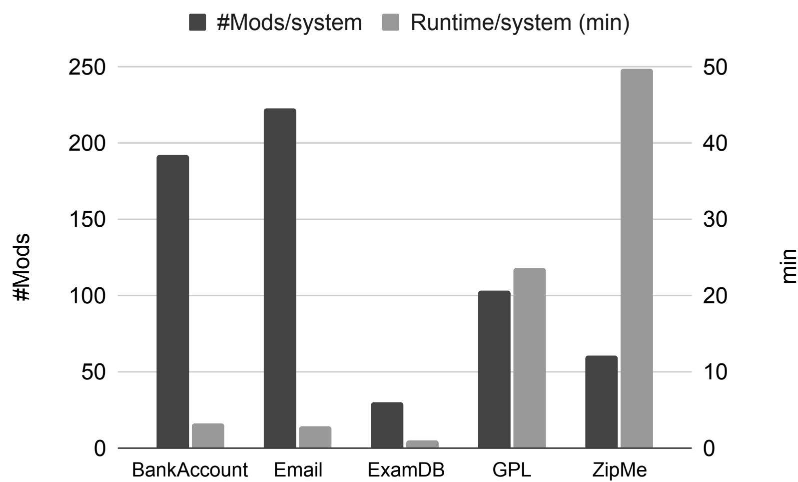
Impact of the characteristics of the different SPL systems

Overall, *the repair performance of* $ProdBased_{enhanced}$ *varies by varying the characteristics of the SPL system under repair.* As shown in Figure 5.5, BankAccount is the system obtaining the highest number of correct patches. The reason is that BankAccount is a small system with multiple similar functions. This could be helpful for redundancy-based APR tools such as jGenProg or Cardumen to find patches in the product under repair itself. Instead, variability bugs in the ExamDB system are fixed by the least number of modification operations. The reason is that the sampled set of this system is small (i.e., 8 products) and only about one or two failing products for each buggy version of this system. Thus, by the product-based approach, the number of failing products of ExamDB system that have been attempted to be fixed is much less than the other systems.

Furthermore, *the running time of the APR approach depends on not only the number of attempted modification operations but also the size of the system and the test suites.* For instance, although each bug in the BankAccount system is attempted by 185 modification operations, which is three times larger than the number of modifications attempted to fix a bug in the ZipMe system, the fixing time of a bug in ZipMe is 15 times larger than that of BankAccount. The reason is that the ZipMe system contains 3,460 statements which is much larger than BankAccount. Also, the test suite of each sampled product of ZipMe is 13 larger than that of a product of BankAccount. As a result, although less modifications are attempted for it, ZipMe still costs more execution time.



(a) Effectiveness (#Plausible fixes, #Correct fixes)

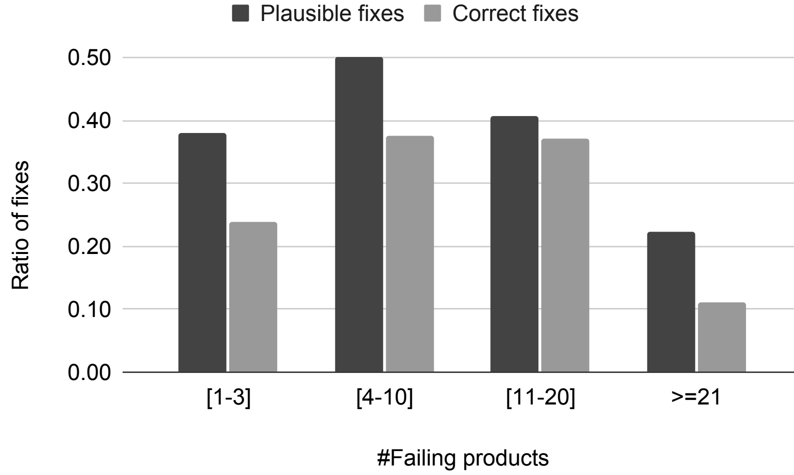


(b) Efficiency (#Mods/system, Runtime/system)

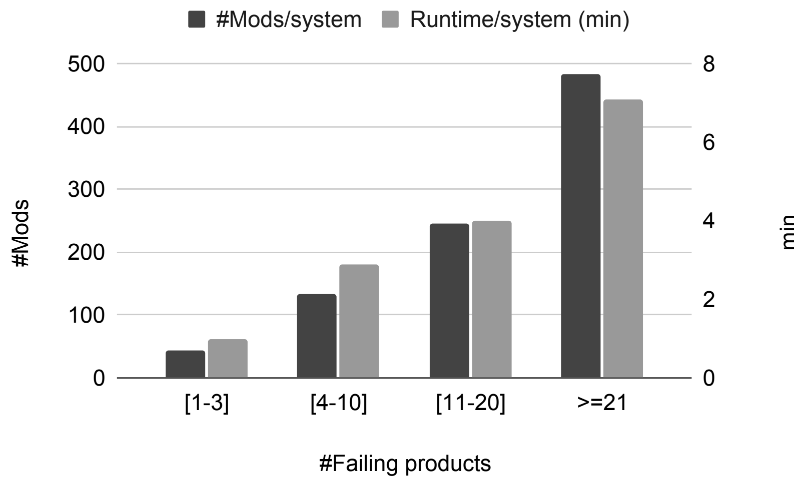
Figure 5.5: RQ3 – The performance of $ProdBased_{enhanced}$ in fixing variability bugs of different SPL systems

Impact of the number of failing products

Figure 5.6 shows the impact of the number of failing products on the effectiveness and efficiency of $ProdBased_{enhanced}$. The performance of $ProdBased_{enhanced}$ could be impacted by various factors such as the nature of code, the size of systems, etc. Thus, to focus on the impact of the number of failing products, this experiment analyzes the performance of $ProdBased_{enhanced}$ on the buggy versions of one system, BankAccount. This experiment separates buggy versions of BankAccount system into different *groups* according to their numbers of failing products (as reported on x-axis of the plots in Figure 5.6).



(a) Effectiveness (#Plausible fixes, #Correct fixes)



(b) Efficiency (#Mods/system, Runtime/system)

Figure 5.6: RQ3 – Impact of the number of failing products on $ProdBased_{enhanced}$'s performance – BankAccount

Moreover, as the numbers of buggy SPL systems in different groups are different, for ease of comparison, this work reports the ratio of fixes (Figure 5.6a), the average number of modification operations, and the average running time (Figure 5.6b) of the bugs in each group. Overall, $ProdBased_{enhanced}$ *obtains the best effectiveness when the bugs cause failures for about 4-20 failing products (12%-60% total sampled products of the system)* (see Figure 5.6a). In addition, *if the number of failing products increases, the number of modification operations and the running time will also increase* (see Figure 5.6b).

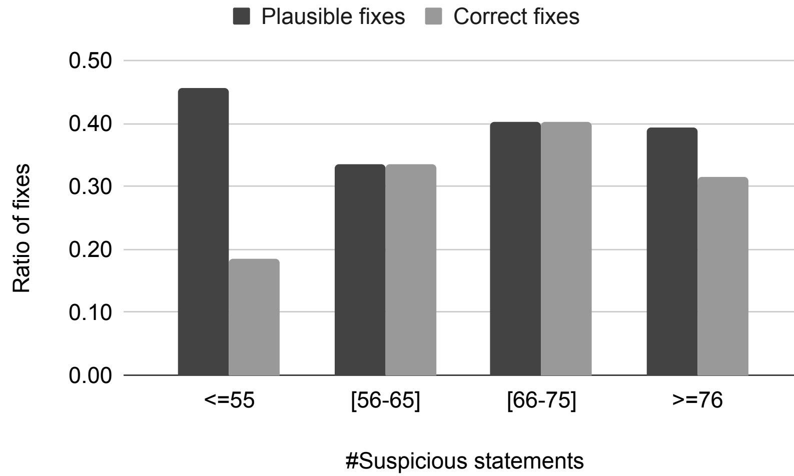
Indeed, for the cases where multiple sampled products are affected by the bugs (i.e.,

the number of failing products is large), the APR tool could need to attempt fixing multiple of them. This increases the attempted modification operations and running time. Intuitively, attempting to repair more products and trying with more modification operations increases the probability of reaching plausible/correct fixes. For the systems containing only one or three failing products, 24% of the bugs are correctly fixed. If the bugs affect about 11-20 products of the system, 37% of the bugs are correctly fixed. However, if there are too many products affected by the bugs, such bugs could be difficult to be fixed. Specifically, if the bugs cause failures for more than 21 products of the system, only 11% of the bugs are correctly fixed.

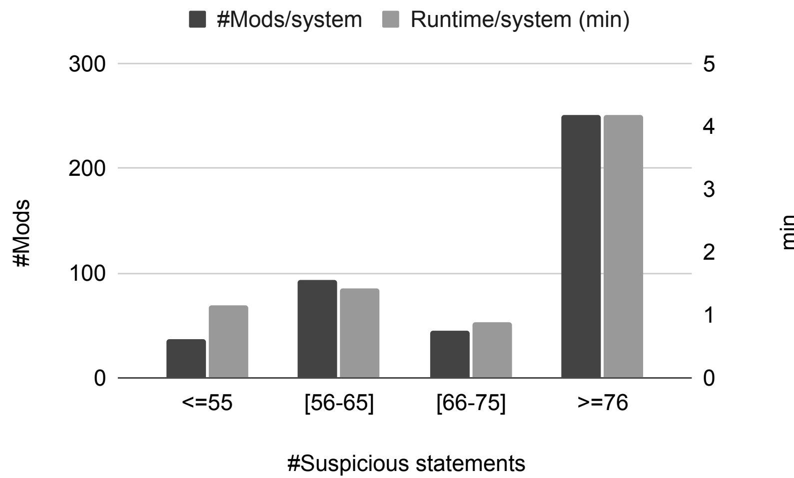
Impact of the number of suspicious statements

Similarly to the previous experiment in Section 5.6.3, to focus on the impact of the number of suspicious statements, this experiment analyzes the repair performance of bugs in one system, BankAccount, and report the ratio of fixes. Figure 5.7 shows how $ProdBased_{enhanced}$'s performance is impacted by the size of suspicious statements. In general, *the number of modification operations and running time increases proportionally with the increase of the number of suspicious statements.* (see Figure 5.7b). Specifically, if there are less than 55 suspicious statements, 36 modification operations are attempted to fix a bug. Instead, if the number of suspicious statements increases 1.5 times, i.e., more than 76 suspicious statements, about 251 modifications are attempted to repair a bug.

The ratio of plausible fixes decreases, yet the ratio of correct fixes increases when the size of suspicious space increases (see Figure 5.7a). For systems with less than 55 suspicious statements, 45% of the bugs are plausibly fixed, and 18% of them are correctly fixed. Instead, if the SPL system has more than 76 suspicious statements, the values of these metrics are 39% and 31%, respectively. This result shows that with a small number of suspicious statements, the APR tool has a high possibility of selecting the buggy statements correctly. Then, it can quickly find a plausible patch and stop searching (high ratio of plausible fixes, yet low ratio of correct fixes). However, with a larger set of suspicious statements, by multiple attempts, $ProdBased_{enhanced}$ has more evidence from previous attempts to find correct modification points and suitable modifications. This could help to increase the ratio of correct fixes. Note that the search process could be much more complicated (i.e., a large amount of modification operations have been attempted) when the search space is too large. This can also negatively impact the



(a) Effectiveness (#Plausible fixes, #Correct fixes)



(b) Efficiency (#Mods/system, Runtime/system)

Figure 5.7: RQ3 – Impact of the number of suspicious statements on $ProdBased_{enhanced}$'s performance – BankAccount

repair performance; indeed, the ratio of correct fixes declines from 40% to 31% when the suspicious statements set increases from 66-75 statements to more than 76 statements.

5.6.4 Threats to Validity

The main threats to the validity of the proposed approaches are: internal, construct, and external validity threats [172].

Threats to internal validity mainly lie in the correctness of the implementation of the proposed approaches. To reduce this threat, the proposed approaches were implemented

on top of a popular APR framework, Astor [48]. The source code was also carefully reviewed and made public so that other researchers can double-check and reproduce the experiments.

Threats to construct validity mainly lie in the rationality of the assessment metrics. To reduce this threat, the popular metrics that have been widely used in previous studies [38, 61, 159, 160, 164, 169–171] were chosen for evaluating both effectiveness and efficiency of the approaches.

Threats to external validity mainly lie in the benchmark used in the experiments. There is only one dataset of artificial bugs is used in the experiments. Thus, the obtained results on artificial faults can not be generalized for large-scale SPL systems containing real faults. To mitigate the threat, five SPL systems which target different application domains and widely-used in existing studies [13, 16, 155], were chosen. In future work, I am planning to collect more real-world variability bugs in larger SPL systems to evaluate the proposed approaches to address these threats. As another external threat, all systems in the benchmark are developed in Java. Therefore, it cannot claim that similar results would have been observed in other programming languages. This is a common threat of several studies on configurable software systems [83, 156]. Another threat lies in the selected APR tools and experiments in only single-bug systems. To reduce the threat, this study chose two representative APR tools targeting at different levels, and they are widely used in the existing studies [61, 161, 164–166]. In the future, I also plan to conduct experiments with more APR tools and multiple-bug cases.

5.7 Summary

Although the bugs in SPL systems could cause severe damage in multiple products, automatically addressing such bugs has not been investigated thoroughly. This chapter introduces two approaches, product-based and system-based, for fixing variability bugs in SPL systems. For the product-based adaptation method, each failing product is fixed individually, and the obtained patches are propagated and validated on the other products of the system. For system-based, the whole SPL system is considered for repair at the same time, i.e., the APR tool is employed to repair the whole system in each iteration. In addition, to improve the performance of APR tools in *navigating modification points* and *selecting suitable modifications*, this chapter proposes several heuristic rules leveraging intermediate validation results of the repaired programs. The experimental results on a

dataset of 318 variability bugs of 5 popular SPL systems show that the product-based adaptation method is better than the system-based adaptation method about 20 times in the number of correct fixes. Notably, the heuristic rules could improve the performance of both adaption methods by 30-150% in the number of correct fixes and 30-50% in the number of attempted modification operations.

Chapter 6

Conclusion

The contribution of the dissertation: SPL systems have gained momentum in the software industry. By the configurable mechanism and reusable parts, SPL engineering allows developers to quickly and easily create multiple products tailored to individual customers' requirements. This helps to reduce costs and improve the performance of the software development process. However, due to the variability inherent to SPL systems, testing and debugging these systems is very challenging. Although automated debugging in single-system engineering has been studied in-depth, debugging SPL systems remain primarily unexplored.

This dissertation aims to shed light on automated debugging SPL systems by focusing on three tasks: *false-passing* product detection, variability fault localization, and variability fault repair. The contributions of the dissertation can be concluded as follows:

First, *the dissertation proposed* CLAP*, an approach for detecting false-passing products of buggy SPL systems.* Chapter 3 formulated the *false-passing* products detection problem. To solve this problem, CLAP proposed six *measurable attributes* to assess the strength of the failure indications in the products (Section 3.3). These indications refer to the *implementation* and *test quality*; the stronger the indications, the more likely the product is *false-passing*.

The experimental results show that CLAP achieves up to 96% *Precision* in detecting *false-passing* products. This means, among 10 products predicted as *false-passing* products by CLAP, there are more than 9 products which are indeed *false-passing* ones. Chapter 3 also evaluates the capability of CLAP in mitigating the negative impact of *false-passing* products on the FL performance. The experiments were conducted on two state-of-the-art variability fault localization approaches with the five most popular SBFL ranking metrics [25, 92]. Interestingly, CLAP can significantly improve their performance in ranking buggy statements by up to 30%. This shows that CLAP can greatly mitigate the negative impact of *false-passing* products on localizing variability bugs and help developers find bugs much faster. The tool is made public at *https://ttrangnguyen.github.io/CLAP/*.

Second, *the dissertation proposed* VarCop*, an approach for localizing variability faults.* Section 4.2 presented the observations about the visibility/invisibility of this kind of fault in SPL systems. Chapter 4 also formulated the conditions (*Buggy PC*) to make variability fault visible in the products of a buggy SPL system and introduced important properties for detecting *Buggy PC*. For a buggy SPL system, VarCop localizes variability bugs by detecting *Buggy PC* to narrow the search space. Then, VarCop considers both the overall and detailed test results to figure out the positions of the faults.

The experimental results show that VarCop significantly outperformed the baselines in all the studied metrics. For the cases containing a single incorrect statement (single-bug), the experimental results show that VarCop significantly outperformed S-SBFL, SBFL, and Arrieta et al. [6] in **all 30/30** metrics by **33%**, **50%**, and **95%** in *Rank*, respectively. Impressively, VarCop correctly ranked the bugs at the top-3 positions in **+65%** of the cases. In addition, VarCop effectively ranked the buggy statements first in about **30%** of the cases, which **doubles** the corresponding figure of SBFL.

For localizing multiple incorrect statements (multiple-bug), after inspecting the first statement in the ranked list resulted by VarCop, up to **10%** of the bugs in a system can be found, which is **2 times** and **10 times** better than S-SBFL and SBFL, respectively. Especially, the experimental results also show that in **22**% and **65%** of the cases, VarCop effectively localized at least one buggy statement of a system at top-1 and top-5 positions. From that, developers can iterate the process of bugs detecting, bugs fixing, and regression testing to quickly fix all the bugs and assure the quality of SPL systems. The tool is made public at ***https://ttrangnguyen.github.io/VARCOP/***.

Third, *the dissertation proposed product-based and system-based approaches for automatically repairing variability faults.* Section 5.3 introduced the detailed algorithms of these two approaches, and their enhanced versions with embedded heuristic rules were introduced in Section 5.4. To improve fault repair performance, the heuristic rules leverage the intermediate repair information to guide the process of navigating modification points and selecting suitable modifications.

The experimental results show that the product-based approach is considerably better than the system-based approach by **12 to 30 times** in the number of plausible fixes and about **20 times** in the number of correct fixes. Interestingly, the heuristics could help to boost the performance of both product-based and system-based approaches by up to **200%**. For instance, by adopting the APR tool Cardumen [63],

$ProdBased_{basic}$ and $SysBased_{basic}$ can **correctly fix 13 and 0 systems** respectively, while $ProdBased_{enhanced}$ and $SysBased_{enhanced}$ **correctly fix 40 and 1 systems** respectively. Moreover, the repair performance could be negatively impacted by FL tools since the modification points are selected based on FL results which are often imperfect. To mitigate the impact of the third-party FL tool, the effectiveness of the repair approaches is assessed when the correct FL results are provided. The experiment results show that the product-based approach is better than the system-based approach about **3 times** in effectiveness and **9 times** in efficiency. In addition, the proposed heuristic rules help to increase **30-150%** the number of correct fixes and decrease **30-70%** the number of attempted modification operations of the corresponding basic approaches. The tool is made public at *https://github.com/ttrangnguyen/SPLRepair*.

The limitation of the dissertation: For each proposed approach, this dissertation carefully analyzes the contribution of each component in the approaches to the whole performance, as well as the sensitivity of the approaches with the different inputs to figure out their weaknesses. Some limitations can be mentioned:

- Although the dataset uses the systems widely used in the existing work, this dataset only contains artificial bugs of Java SPL systems, so the similar results cannot be concluded for real-world faults.
- All systems in the benchmark are developed in Java. Therefore, it cannot claim that similar results would have been observed in other programming languages or technologies.
- To guarantee the reliability of SPL systems' test results, the flaky test problem is still challenging and has not been addressed.

Future works: From the results achieved in the dissertation, as well as the remaining limitations, there are some research directions for future work:

- *Collecting real-world variability bugs in larger SPL systems to more thoroughly evaluate the techniques.* Abal et al. [76] have collected and presented a dataset of 98 real-world variability bugs in Linux, Apache, BusyBox, and Marlin systems. These bugs are essential for evaluating the QA tools of SPL systems. However, most of these bugs are compilation bugs, and they are not provided with test suites. Thus, this dataset does not fit well with the approaches leveraging testing information

like SBFL, Clap, or VarCop, etc. In practice, collecting real-world bugs is very challenging. Thus, it requires in-depth analysis and design to collect the bug systematically and automatically. In future work, I plan to investigate the bug-fixing commits which are often logged and reported. These commits could be used to trace back to find the bug-introducing commits and then the buggy versions of the systems can be obtained.

- *Extending the experiments with more APR tools.* Section 5.6 evaluates the variability bug repair performance with jGenProg and Cardumen. These tools can repair the program at different levels, i.e., statement and expression levels. However, there are much more APR tools, especially with the development of large language models and generative AI, multiple new APR tools have been introduced. In the next study, I plan to conduct more experiments with diverse APR tools to thoroughly evaluate the contribution of the heuristic rules and extend the conclusions.

- *Handling the flaky test problem to improve the quality of the test suites.* For the debugging approaches leveraging test results, the quality of the test suites is an essential factor. The low-quality test suites could result in both coincidental correctness and flaky test problems. The coincidental correctness leads to under-counting the failed tests and over-counting the passed tests, negatively impacting the performance of FL approaches. In this dissertation, Clap has been introduced to address this phenomenon at the product level. Meanwhile, the flaky tests yield both passing and failing results despite zero changes to the code or test [173]. This unreliability of the test results provides incorrect indications for FL and APR techniques. Thus diminishing their performance. In the future, I plan to analyze the symptoms of the flaky tests and design a specialized approach to detect these tests in SPL systems.

List of Publications

NTT1 . Nguyen, Thu-Trang, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. "A variability fault localization approach for software product lines." IEEE Transactions on Software Engineering 48, no. 10 (2021): ISSN 0098-5589, DOI: `https://doi.org/10.1109/TSE.2021.3113859`, ISI/Q1.

NTT2 . Nguyen, Thu-Trang, and Hieu Dinh Vo. "Detecting Coincidental Correctness and Mitigating Its Impacts on Localizing Variability Faults." In 2022 14th International Conference on Knowledge and Systems Engineering (KSE), pp. 1-6. IEEE, 2022.

NTT3 . Nguyen, Thu-Trang, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. "Detecting false-passing products and mitigating their impact on variability fault localization in software product lines." Information and Software Technology 153 (2023): ISSN 0950-5849, volume 153, DOI: `https://doi.org/10.1016/j.infsof.2022.107080`, ISI/Q1.

NTT4 . Nguyen, Thu-Trang, Xiao-Yi Zhang, Paolo Arcaini, Fuyuki Ishikawa, and Hieu Dinh Vo. "Automated Program Repair for Variability Bugs in Software Product Line Systems." Journal of Systems and Software. ISI/Q1 (accepted).

This list contains four publications.

Bibliography

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated, 2013.

[2] Ivan do Carmo Machado, John D McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, 2014.

[3] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 643–654. IEEE, 2016.

[4] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[5] Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Variability fault localization: A benchmark. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, page 120–125, 2021.

[6] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Spectrum-based fault localization in software product lines. *Information and Software Technology*, 100:18–31, 2018.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. Software product lines. *Feature-Oriented Software Product Lines: Concepts and Implementation*, pages 3–15, 2013.

[8] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software*, 182:111044, 2021.

[9] Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, and Mathieu Acher. On the benefits and limits of incremental build of software configurations: An exploratory study. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1584–1596, New York, NY, USA, 2022. Association for Computing Machinery.

[10] David M Weiss. The product line hall of fame. In *SPLC*, volume 8, page 39, 2008.

[11] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines.* Springer, 2016.

[12] Brady J Garvin and Myra B Cohen. Feature interaction faults revisited: An exploratory study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 90–99. IEEE, 2011.

[13] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 421–432, 2014.

[15] Son Nguyen, Hoan Nguyen, Ngoc Tran, Hieu Tran, and Tien Nguyen. Feature-interaction aware configuration prioritization for configurable code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 489–501. IEEE, 2019.

[16] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 483–494, 2016.

[17] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[19] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55, 2012.

[20] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, pages 1–5, 2011.

[21] Wes Masri and Rawad Abou Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM transactions on software engineering and methodology (TOSEM)*, 23(1):1–28, 2014.

[22] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. Killing stubborn mutants with symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–23, 2021.

[23] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. How test suites impact fault localisation starting from the size. *IET software*, 12(3):190–205, 2018.

[24] Weibo Wang, Yonghao Wu, and Yong Liu. A passed test case cluster method to improve fault localization. *Journal of Circuits, Systems and Computers*, 30(03):2150053, 2021.

[25] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[26] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[27] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[28] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on

a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.

[29] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[30] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[31] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[32] Ivan do Carmo Machado, John D McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–8, 2012.

[33] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[34] Maggie Hamill and Katerina Goseva-Popstojanova. Analyzing and predicting effort associated with finding and fixing software faults. *Information and Software Technology*, 87:1–18, 2017.

[35] Jorge Echeverría, Francisca Pérez, Andrés Abellanas, Jose Ignacio Panach, Carlos Cetina, and Óscar Pastor. Evaluating bug-fixing in software product lines: An industrial case study. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 441–444, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55, jan 2017.

[38] Yi Li, Shaohua Wang, and Tien N. Nguyen. DEAR: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 511–523, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.*, 45(1):34–67, jan 2019.

[41] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automatic detection and removal of conformance faults in feature models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 102–112, April 2016.

[42] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automated repairing of variability models. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 9–18, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 625–636. IEEE Press, 2017.

[44] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. Range fixes: Interactive error resolution for software configuration. *Ieee transactions on software engineering*, 41(6):603–619, 2014.

[45] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

[46] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[47] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

[48] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *J. Syst. Softw.*, 151(C):65–80, may 2019.

[49] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 75–87, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. TransplantFix: Graph differencing-based code transplantation for automated program repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *2010 Third international conference on software testing, verification and validation*, pages 459–468. IEEE, 2010.

[52] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *International Conference on Software Product Lines*, pages 196–210. Springer, 2010.

[53] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55, 2012.

[54] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000#

ifdefs issue. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 421–432, 2014.

[55] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 197–206, 2014.

[56] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling*, 18:499–521, 2019.

[57] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022.

[58] Akira Ochiai. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. *Bulletin of Japanese Society of Scientific Fisheries*, 22:526–530, 1957.

[59] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 2019.

[60] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[61] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 615–627, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. Association for Computing Machinery.

[63] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In Thelma Elita Colanzi and Phil McMinn, editors, *Search-Based Software Engineering*, pages 65–86, Cham, 2018. Springer International Publishing.

[64] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13. IEEE Press, 2012.

[65] He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Softw. Engg.*, 26(2), mar 2021.

[66] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. Association for Computing Machinery.

[68] Jiajun Jiang, Yingfei Xiong, and Xin Xia. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science china information sciences*, 62:1–16, 2019.

[69] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8, 2013.

[70] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.

[71] Sven Apel, Alexander Von Rhein, Thomas ThüM, and Christian KäStner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 2013.

[72] Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of ltl properties—a case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[73] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. Configuration-aware change impact analysis (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–395. IEEE, 2015.

[74] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 59–66, 2018.

[75] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S Foster, and Adam A Porter. igen: Dynamic interaction inference for configurable software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 655–665, 2016.

[76] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–34, 2018.

[77] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[78] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. Empirical analysis of the tool support for software product lines. *Software and Systems Modeling*, 22(1):377–414, 2023.

[79] Chin Khor and Robyn R Lutz. Requirements analysis of variability constraints in a configurable flight software system. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 244–254. IEEE, 2023.

[80] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):1–39, 2012.

[81] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91, 2013.

[82] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. Exploring feature interactions without specifications: A controlled experiment. *ACM SIGPLAN Notices*, 53(9):40–52, 2018.

[83] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. Faster variational execution with transparent bytecode transformation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[84] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. *ACM SIGPLAN Notices*, 48(6):355–364, 2013.

[85] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. Hinnperf: Hierarchical interaction neural network for performance prediction of configurable systems. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–30, 2023.

[86] CKJDSA Stefan Mühlbauer, Florian Sattler, and N Siegmund. Analyzing the impact of workloads on modeling the performance of configurable software systems. In *Proceedings of the International Conference on Software Engineering (ICSE), IEEE*, 2023.

[87] Rabatul Aduni Sulaiman, Dayang NA Jawawi, and Shahliza Abdul Halim. Cost-effective test case generation with the hyper-heuristic for software product line testing. *Advances in Engineering Software*, 175:103335, 2023.

[88] Mutlu Beyazit, Tugkan Tuglular, and Dilek Öztürk Kaya. Incremental testing in software product lines—an event based approach. *IEEE Access*, 11:2384–2395, 2023.

[89] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[90] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pair-wise testing for software product lines. In *Proceedings of the 17th international software product line conference*, pages 227–235, 2013.

[91] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 421–432, 2014.

[92] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.

[93] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pair-wise testing of software product lines. In *International Conference on Software Product Lines*, pages 196–210. Springer, 2010.

[94] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 244–254. IEEE, 2012.

[95] Son Nguyen. Feature-interaction aware configuration prioritization. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 974–976, 2018.

[96] Isis Cabral, Myra B Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *International Conference on Software Product Lines*, pages 241–255. Springer, 2010.

[97] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, 2014.

[98] Alexandre Perez, Rui Abreu, and Arie Van Deursen. A theoretical and empirical analysis of program spectra diagnosability. *IEEE Transactions on Software Engineering*, 2019.

[99] Alberto Gonzalez-Sanchez, Hans-Gerhard Gross, and Arjan JC van Gemund. Modeling the diagnostic efficiency of regression test suites. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 634–643. IEEE, 2011.

[100] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, pages 82–91, 2006.

[101] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[102] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[103] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *2009 IEEE 31st International Conference on Software Engineering*, pages 45–55. IEEE, 2009.

[104] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324, 2013.

[105] Zheng Li, Meiying Li, Yong Liu, and Jingyao Geng. Identify coincidental correct test cases based on fuzzy classification. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 72–77. IEEE, 2016.

[106] Xiaozhen Xue, Yulei Pang, and Akbar Siami Namin. Trimming test suites with coincidentally correct test cases for enhancing fault localizations. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 239–244. IEEE, 2014.

[107] Wes Masri and Rawad Abou Assi. Cleansing test suites from coincidental correctness to enhance fault-localization. In *2010 third international conference on software testing, verification and validation*, pages 165–174. IEEE, 2010.

[108] Aritra Bandyopadhyay. Mitigating the effect of coincidental correctness in spectrum based fault localization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 479–482. IEEE, 2012.

[109] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[110] Frank Tip and TB Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):5–55, 2001.

[111] Richard A DeMillo, Hsin Pan, and Eugene H Spafford. Critical slicing for software fault localization. *ACM SIGSOFT Software Engineering Notes*, 21(3):121–134, 1996.

[112] Yue Yan, Shujuan Jiang, Yanmei Zhang, Shenggang Zhang, and Cheng Zhang. A fault localization approach based on fault propagation context. *Information and Software Technology*, 160:107245, 2023.

[113] Attila Szatmári. Towards context-aware spectrum-based fault localization. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 496–498. IEEE, 2023.

[114] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 520–523. IEEE, 2011.

[115] James S Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *Managing Requirements Knowledge, International Workshop on*, pages 539–539, 1987.

[116] Hiralal Agrawal, Richard A. De Millo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[117] Bogdan Korel. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.

[118] W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[119] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272, 2017.

[120] Nazanin Bayati Chaleshtari and Saeed Parsa. Smbfl: slice-based cost reduction of mutation-based fault localization. *Empirical Software Engineering*, 25(5):4282–4314, 2020.

[121] Xiangyu Li and Alessandro Orso. More accurate dynamic slicing for better supporting software debugging. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 28–38. IEEE, 2020.

[122] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, nov 2019.

[123] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. Association for Computing Machinery.

[124] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2020.

[125] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 298–309, New York, NY, USA, 2018. Association for Computing Machinery.

[126] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 660–670. IEEE Press, 2017.

[127] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781. IEEE Press, 2013.

[128] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.

[129] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 1345–1351. AAAI Press, 2017.

[130] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pages 1161–1173. IEEE Press, 2021.

[131] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[132] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 50–61, 2019.

[133] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, 2009.

[134] Giovani Guizzo, Federica Sarro, and Mark Harman. Cost measures matter for mutation testing study validity. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1127–1139, 2020.

[135] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[136] W Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[137] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium*

and the 13th European conference on Foundations of software engineering, pages 416–419, 2011.

[138] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[139] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[140] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE, 2008.

[141] Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. Feature-oriented defect prediction. In *Proceedings of the 24th ACM conference on systems and software product line: Volume A-Volume A*, pages 1–12, 2020.

[142] Le Hoang Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Thi Minh Phuong, and Pham Huy Thong. Empirical study of software defect prediction: a systematic mapping. *Symmetry*, 11(2):212, 2019.

[143] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[144] Max Kuhn, Kjell Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013.

[145] Ling Bian and Rachel Butler. Comparing effects of aggregation methods on statistical and spatial properties of simulated spatial data. *Photogrammetric engineering and remote sensing*, 65:73–84, 1999.

[146] H Ding, L Chen, J Qian, L Xu, and BW Xu. Fault localization method using information quantity. *Ruan Jian Xue Bao/Journal of Software*, 24(7):1484–1494, 2013.

[147] Lucia Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 127–138, 2014.

[148] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[149] Andréia da Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira de Souza, and Cláudio Lopes de Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[150] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In *Proceedings of the 28th international conference on Software engineering*, pages 827–830, 2006.

[151] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018.

[152] Ben H Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical software engineering*, 14(3):341–369, 2009.

[153] Joseph L Fleiss. Estimating the accuracy of dichotomous judgments. *Psychometrika*, 30(4):469–479, 1965.

[154] Fernando Lourenço, Victor Lobo, and Fernando Bacao. Binary-based similarity measures for categorical data and their application in self-organizing maps. *JOCLAD 2004-XI Jornadas de Classificacao e Anlise de Dados*, pages 1–8, 2004.

[155] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2011.

[156] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. Balancing soundness and efficiency for practical testing of configurable systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 632–642. IEEE, 2017.

[157] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants:

Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014.

[158] Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.

[159] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 19–30, New York, NY, USA, 2019. Association for Computing Machinery.

[160] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[161] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. Evaluating and improving unified debugging. *IEEE Transactions on Software Engineering*, 48(11):4692–4716, 2021.

[162] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. Utilizing source code embeddings to identify correct patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 18–25, 2020.

[163] Jinhan Kim, Jongchan Park, and Shin Yoo. The inversive relationship between bugs and patches: An empirical study. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 314–323. IEEE, 2023.

[164] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. 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, pages 302–313, New York, NY, USA, 2019. Association for Computing Machinery.

[165] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171:110825, 2021.

[166] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 968–980, New York, NY, USA, 2021. Association for Computing Machinery.

[167] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662, 2018.

[168] Patrick E McKnight and Julius Najab. Mann-whitney U test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[169] Yuan Yuan and Wolfgang Banzhaf. Toward better evolutionary program repair: An integrated approach. *ACM Trans. Softw. Eng. Methodol.*, 29(1), jan 2020.

[170] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1506–1518, New York, NY, USA, 2022. Association for Computing Machinery.

[171] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129, 2018.

[172] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated, 2012.

[173] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(3):72, 2023.