



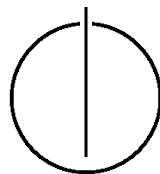
FAKULTÄT FÜR INFORMATIK

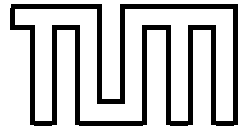
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

A Framework for Failure Diagnosis

Mojdeh Golagha





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl IV - Software and Systems Engineering

A Framework for Failure Diagnosis

Mojdeh Golagha

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Thomas Neumann

Prüfer der Dissertation:

1. Prof. Dr. Alexander Pretschner
2. Prof. Lionel Briand, Ph.D.,
University of Luxembourg, Luxembourg

Die Dissertation wurde am 19.08.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.01.2020 angenommen.

Acknowledgments

Undertaking this Ph.D. has been a truly life-changing experience for me, and it would not have been possible to do without the support and guidance that I received from many people, and I would not forget to thank them.

My Supervisor: Alex, you decided to trust your instinct. Your decision changed a lot of things in my life. I worked hard to prove your instinct was right. Throughout this journey, under your supervision, I grew and became the woman I am today. Thank you for helping me understanding my strengths and weaknesses, teaching me how to accept them and how to use them. Thank you for showing me how to admit and say sorry when I am wrong, how to argue and convince when I am right, and how to confidently say "I don't know but I'll find out". I learned more than you know from you. I am more than grateful.

My second supervisor: Lionel, I would like to express my sincere gratitude. Thank you for hosting me at SnT center and providing useful and actionable feedback. I was inspired by many of your previous works. My short stay with your research group and interactions with colleagues there provided me new insight.

My past and present colleagues: Traudl, Florian, Dominik, Matthias, Benni, Tobias, Kristian, Prachi, Enrico, Flo, Ehsan, Ana, Thomas, Markus, Valentin, Tabea and Daniel, thank you for all the collaborations, discussions, cake and fun we had together. Patrick, Haafiz, and Ilias thank you for reading parts of this thesis and helping me to improve it. Alei, Amjad, Mohsen, Saahil, Severin, and Sebastian, thank you for all the CBs, laughter, discussions, and nights out. Daniel, Constantin, David, Rezaur, Mohammed, and all other master's, bachelor's students, and Hiwis, thank you for choosing me, helping me, and all your hard work that contributed to this thesis.

My friends: Tannaz, you know without you a lot of days were unbearable. Thank you for always being there for me, listening to me, encouraging me and telling me that I can do it. Sara, Parisa, Homa, thank you for all the chats, girls nights out and the support you have given me. The "sexiest scientist" title will continue to bring me a big smile forever!

My family: Mom, Dad, whatever I have and whatever I am today is because of you. Thank you for all your unconditional love and support and for pushing me to start this journey. Mehrdad, Johanna, without you I couldn't have done it. I will never forget what you did for me. Mahshid, thank you for being my little sister I guess, and all the love you give me that makes me buy you the most expensive gifts I've ever bought for anybody.

My Navid, thank you for holding my hand and heart throughout all ups and downs of this journey. Thank you for reminding me every day why I should cherish every moment of it. Watching you willing, learning, and trying has made me a better person.

Zusammenfassung

Einer der teuersten und herausforderndsten Abschnitte im Softwareentwicklungszyklus ist das Debugging. Vor allem die Zeit, die während des Debugging-Prozesses benötigt wird, um einen Failure zu analysieren und den zugrunde liegenden Fault zu finden, ist ein wichtiger Kostenfaktor. Zwei beispielhafte Verfahrensweisen, die es den Testern ermöglichen können, diese Zeit für die Fehleranalyse zu verkürzen, sind das Failure Clustering und die automatische Fault Lokalisierung. Obwohl in der Literatur eine Vielzahl von Varianten dieser Techniken beschrieben wird, wird ihre Anwendung in realen Umgebungen immer noch von Problemen verhindert. Die Tester werden zudem bei der Auswahl einer geeigneten Methode für ihre Domäne durch diese Vielzahl an verschiedenen Techniken zusätzlich verwirrt.

Normalerweise wird, ungeachtet der Domäne, im Zuge von Regressionen regelmäßig eine große Anzahl von Tests durchgeführt. Kommt es dabei zum Fehlschlagen vieler Tests, ist üblicherweise nur eine geringe Anzahl von zugrundeliegenden Faults dafür verantwortlich. Berücksichtigt man jedoch die extrem lange Ausführungsdauer der Tests, ist es nicht immer möglich, den ursächlichen Fault zu finden und ihn zu entfernen, um dann alle Tests erneut ausführen zu lassen. Da meist im Vorfeld ein enges Zeitfenster für das Finden und Entfernen der Faults festgelegt wurde, stehen die Entwickler zusätzlich unter einem enormen Zeitdruck. Daher braucht es ein Assistenzsystem, welches die Produktivität der Entwickler erhöht und die Zeit für die Failure-Analyse verringert.

Um die eben beschriebenen Diskrepanzen zu überbrücken und den aktuellen Forschungsstand in die Praxis zu integrieren, haben wir ein Framework für Fehlerdiagnose entwickelt. Dieses Framework fungiert dabei als Assistenzsystem für Entwickler, um es diesen zu ermöglichen, schnelle und zuverlässige Failure Diagnosen durchzuführen.

Durch die Benutzung des Frameworks wissen die Entwickler, welche Daten sie benötigen, um weitere Analysen durchzuführen. Sie sind außerdem in der Lage, Failure hinsichtlich ihres Ursprungs zu kategorisieren und können dann in einem weiteren Schritt mehr Informationen über die Ursprünge einer jeden Failure Gruppe erlangen. Ursprünglich wurde unser Framework auf Basis von aktuellen Failure-Diagnosetechniken entwickelt. Dabei implementierten und evaluierten wir die existierenden Techniken und Verfahren. Wo die Ergebnisse vielversprechend waren, knüpften wir an diese an, verbesserten sie und schlugen, wo erforderlich, neue Lösungen vor. Mit unseren Evaluationsproben konnten wir zeigen, dass unser Framework in der Reduzierung der Failure-Analysezeit effektiver ist, als die aktuell in der Forschung zu findenden Methoden es sind.

Abstract

Debugging is one of the most expensive and challenging phases in the software development life-cycle. One important cost factor in the debugging process is the time required to analyze failures and find underlying faults. Two types of techniques that can help testers to reduce this analysis time are Failure Clustering and Automated Fault Localization. Although there is a plethora of these techniques in the literature, there are still some gaps that prevent their operationalization in real-world contexts. Besides, the abundance of these techniques confuses the testers in selecting a suitable method for their specific domain.

Regardless of the domain, there is usually a large number of tests that are frequently executed as regressions happen. In case of large numbers of failing tests, there are usually only a few underlying faults that cause a large number of tests to fail. Considering the significantly high execution time of the tests, it is not always possible to find the first fault, resolve it, and re-run all the tests. In addition, due to having a limited pre-determined time to find and repair faults, developers are usually under a lot of pressure. Therefore, there is a need for an assisting tool to improve developers' productivity and reduce failure analysis time.

To fill the gaps and bring state-of-art closer to practice, we developed a framework for failure diagnosis. This framework would serve as an assisting tool for developers to empower them for fast and reliable failure diagnosis. Utilizing this framework, developers know which data they need for further analysis, are able to group failures based on their root causes, and are able to find more information about the root causes of each failing group.

Our framework was initially developed based on state-of-the-art failure diagnosis techniques. We implemented and evaluated existing techniques. We built on and improved them where the results were promising and proposed new solutions where needed. With our evaluation experiments, we show that our framework is more effective in reducing failure analysis time than state-of-the-art techniques.

Outline of the Thesis

CHAPTER 1: INTRODUCTION

This chapter presents an introduction to the topic and the fundamental issues addressed by this thesis. It discusses ideas, goals, and limitations of this work.

CHAPTER 2: BACKGROUND

This chapter presents an introduction to the failure clustering and fault localization techniques and their advantages in failure diagnosis. Parts of this chapter have previously appeared in publications [55, 58, 54, 56, 150], co-authored by the author of this thesis.

CHAPTER 3: CASE STUDIES

This chapter presents an overview of the case studies used in the evaluations. Parts of this chapter have previously appeared in peer-reviewed publications [150, 54], co-authored by the author of this thesis.

CHAPTER 4: FAILURE CLUSTERING WITH COVERAGE

This chapter presents a clustering approach to group failures based on their root-causes. In addition, it presents a methodology to apply failure clustering in a real-world context. Parts of this chapter have previously appeared in a publication [58], co-authored by the author of this thesis.

CHAPTER 5: FAILURE CLUSTERING WITHOUT COVERAGE

This chapter presents a failure clustering approach that does not need coverage data. Parts of this chapter have previously appeared in a publication [57], co-authored by the author of this thesis.

CHAPTER 6: IMPROVING SBFL THROUGH USING SYNTACTIC BLOCK GRANULARITY

This chapter presents a new granularity level that helps in boosting spectrum-based fault localization effectiveness.

CHAPTER 7: IMPROVING SBFL THROUGH TACKLING CONFOUNDING BIAS

This chapter presents a new ranking technique to improve the effectiveness of spectrum-based fault localization.

CHAPTER 8: PREDICTING THE QUALITY OF SBFL

This chapter presents a model to predict the quality of spectrum-based fault localization. Considering the prediction, developers can decide whether to use the fault localization tool or not.

CHAPTER 9: RELATED WORK

This chapter presents related work in the sub-fields of fault localization and failure clustering. Parts of this chapter have been published in peer-reviewed publications [55, 58, 54, 56, 150] co-authored by the author of this thesis.

CHAPTER 10: CONCLUSIONS

This chapter first presents a summary of what has been done throughout the chapters of this thesis. Subsequently, we state the results of the thesis and the lessons learned during the development of this work. Afterward, we discuss limitations and avenues for future work.

N.B.: Multiple chapters of this dissertation are based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked explicitly.

Contents

Acknowledgements	v
Zusammenfassung	vii
Abstract	ix
Outline of the Thesis	xi
Contents	xiii

I. Introduction and Background	1
1. Introduction	3
1.1. Failure Diagnosis: Benefits, Early Works, and Shortcomings	4
1.1.1. Failure Clustering	4
1.1.2. Fault Localization	5
1.2. Goal	7
1.3. Problem Statement and Research Questions	7
1.4. Solution	9
1.5. Contributions	10
1.6. Summary of Results	13
1.7. Structure	13
2. Background	15
2.1. Failure Clustering Background	15
2.1.1. Agglomerative Hierarchical Clustering	15
2.1.2. Evaluation Metrics	18
2.2. Fault Localization Background	21
2.2.1. Spectrum-based Fault localization	21
2.2.2. Method-Level Fault Localization with Causal Inference	22
2.2.3. Evaluation Metrics	23
3. Case Studies	25
3.1. Large Car Manufacturer (LCM)	25
3.2. Defects4J	30

3.3. Pairika	31
II. Failure Clustering	33
4. Failure Clustering with Coverage	35
4.1. Introduction	36
4.2. Methodology	36
4.2.1. Approach	37
4.2.2. Parameter Setting	40
4.3. Experiment	42
4.3.1. Research Questions	42
4.3.2. Results	43
4.3.3. Threats to Validity	45
4.4. Conclusion	47
5. Failure Clustering without Coverage	49
5.1. Introduction	49
5.2. Approach	50
5.2.1. Generating the Input Data	51
5.2.2. Clustering Failing Tests	55
5.2.3. Selecting the Representatives	56
5.3. Experiment	56
5.3.1. Parameter Setting	57
5.4. Evaluation	58
5.4.1. Summary of Results	58
5.4.2. Threats to Validity	61
5.5. Conclusion	62
III. Fault Localization	63
6. Improving SBFL Through Using Syntactic Block Granularity	65
6.1. Introduction	66
6.2. Motivation	66
6.3. Syntactic Block Granularity	70
6.4. Experiments	71
6.4.1. Data set	72
6.4.2. Implementation	72
6.4.3. Research Questions	74
6.5. Results	75
6.5.1. Evaluation Results	75

6.5.2. Threats to Validity	81
6.6. Conclusion	81
7. Improving SBFL Through Tackling Confounding Bias	83
7.1. Introduction	83
7.2. Methodology	86
7.2.1. Approach	86
7.2.2. Parameter setting	89
7.3. Experiment	91
7.3.1. Research Questions	91
7.4. Results	91
7.4.1. Evaluation Results	91
7.4.2. Threats to Validity	95
7.5. Practical Implications	96
7.5.1. Triggering a method after test failure	96
7.5.2. Combining unit and integration test suites	97
7.6. Conclusion	98
8. Predicting the Quality of SBFL	99
8.1. Introduction	100
8.2. Methodology	101
8.3. Generating the Data Set	102
8.3.1. Variables	102
8.3.2. Target Class	104
8.3.3. Populating the Data Set	105
8.4. Data Analysis	105
8.4.1. Pearson’s Correlation	105
8.4.2. Logistic Regression	106
8.5. Summary of Results	114
8.5.1. Research Questions	114
8.5.2. Threats to Validity	118
8.6. Conclusion	118
 IV. Related Work and Conclusion	 119
9. Related Work	121
9.1. Case Study	121
9.2. Failure Clustering	122
9.3. Fault Localization	124
9.3.1. Spectrum-based Fault Localization	124
9.3.2. Fault localization with causal inference	126

9.4. Failure Diagnosis Framework	126
9.5. Predicting the Quality of Fault Localization	127
10. Conclusions	129
10.1. Aletheia: Failure Clustering and Fault Localization in a Pipeline	129
10.1.1. Data Generation	129
10.1.2. Failure Clustering	131
10.1.3. Fault Localization	131
10.1.4. Evaluation	131
10.2. Summary	133
10.3. Results and Lessons Learned	134
10.4. Limitations	134
10.5. Future Work	135
Bibliography	139
Index	157
List of Figures	157
List of Tables	159

Part I.

Introduction and Background

1. Introduction

This chapter presents an introduction to the topic and the fundamental issues addressed by this thesis. It discusses ideas, goals, and limitations of this work.

Software debugging has been recognized to be time-consuming, tiresome and expensive. Software failures cost billions for economies every year [173]. Even when the existence of faults in software is discovered due to failures, finding the causes to repair them is an entirely different matter. In our terminology, a *failure* is the deviation of actual run-time behavior from intended behavior, and a *fault* is the reason for the deviation [175]¹. In other words, a fault is the program element that needs to be changed in order to remove the failure.

The essence of failure diagnosis is to trace back from a failure to the fault or faults [65]. Automated diagnosis can reduce the effort spent on manual debugging, which shortens the test-diagnose-repair cycle, and can therefore be expected to lead to more reliable systems, and a shorter time-to-market. In this thesis, we focus on automated diagnosis techniques and explain our endeavor in improving them to make them applicable in practice.

We recognized two general categories of techniques for addressing reducing failure diagnosis time: *failure clustering* and *automated fault localization*.

Failure clustering methods attempt to group failing tests with respect to the faults that caused them [147]. If there are several failing test cases (TC) as the result of test execution, these failing TCs may be clustered such that tests which are in the same cluster would have failed due to the same hypothesized fault. Then, in an ideal world, testers investigate only one representative TC from each cluster to discover all the underlying faults. This process eliminates the need for analyzing each failing TC individually. Thus, there would be a significant reduction in analysis time[58].

Automated fault localization techniques aim to “identify some program event(s) or state(s) or element(s) that cause or correlate with the failure to provide the developer with a report that will aid fault repair” [119]. The debugging process is usually predicated on the developer’s ability to find and repair faults. While both steps in the debugging process (fault localization and fault repair) are time-consuming in their own right, fault localization is considered more critical, as it is a prerequisite for fault repair [119]. Furthermore, Kochhar et al. have found that there is a large demand for fault localization solutions among developers [93]. Therefore, over the past ten years, a lot of research has gone into developing automated techniques for fault localization in order to help speed up the process [182].

¹In this thesis, we use “fault”, “root cause”, and “bug” terms interchangeably.

1.1. Failure Diagnosis: Benefits, Early Works, and Shortcomings

There is a plethora of failure clustering and automated fault localization techniques in the literature (see Chapter 9). Although developers find these methods worthwhile and essential [94], these techniques are not adopted in practice yet. There are gaps in need of filling to make these methods applicable. In the following, we introduce these techniques and their shortcomings.

1.1.1. Failure Clustering

Clustering failures is effective in reducing failure analysis time [58]. Its advantages are threefold:

1. It eliminates the need to analyze each failing test individually. To achieve this goal, it is enough to select one representative for each cluster and analyze only the representatives to find all the underlying faults in case of multiple faults [58].

In an industrial environment for embedded systems, usually several dozens of TCs are executed each night as regressions happen, and several hundred every weekend, which together usually lead to large numbers of failures. Developers must analyze the failing tests and find all the root causes in the short time they have before the software release. The complexity of the analysis process makes the failure diagnosis process tough and time-consuming. Since in practice, a single fault usually leads to the failure of multiple TCs, analyzing only the representative TCs help developers to find more faults in a shorter time.

2. It provides the opportunity for debugging in parallel [77].
3. It gives an estimation of the number of faults causing the failures. Fault localization while there are several faults in the code, is more challenging than when there is only one fault in the code. When a program fails, the number of faults is, in general, unknown; and certain faults may mask or obfuscate other faults [77].

Jones et al. [77] introduced a parallel debugging process as an alternative to sequential debugging. They suggest that in the presence of multiple faults in a program, clustering failing tests based on their underlying faults, and assigning clusters to different developers for simultaneous debugging, reduce the total debugging cost and time. They propose two clustering techniques. Using the first technique, they cluster failures based on execution profiles and fault localization results. They start the clustering process by using execution profile similarities and complete it using fault localization results. Their second technique suggests to only use the results of fault localization.

Hoegerle et al. [68] introduced another parallel debugging method which is based on integer linear programming [134]. They applied the above-mentioned second clustering technique of Jones et al. to compare it with their own debugging approach. Their results

show that this clustering technique of Jones et al. is not so effective. But, the first technique is effective if it is adapted to the context.

Parallel debugging reduces the analysis time. However, it does not remove the need for analyzing all the failing tests one by one. It provides segregation between faults to make them more localizable. But, it does not provide segregation between failing tests.

Another shortcoming in this area of research is the lack of a methodology for adapting this idea to different industrial domains.

Moreover, the other similar existing approaches in the literature (see Chapter 9) are either based on coverage data or use context-specific data [153]. Therefore, there is a need for other sources of non-coverage data for the cases that the source code or execution profile is not available.

1.1.2. Fault Localization

Finding and fixing faults requires developers to put in a significant amount of time and effort. Fault Localization (FL) is a process to find the location of faults in a program. To reduce the developers workload and debugging cost, researchers have studied different approaches to automate this process [182]. Due to high demand, there have been many attempts to develop useful FL tools in recent years.

One of the most popular subsets of automated FL techniques is spectrum-based fault localization, known as SBFL [182]. In order to correlate program elements with failing TCs, these techniques are built upon abstractions of program execution traces, also known as program spectra, executable statement hit spectra, or code coverage [182]. These program spectra can be defined as a set of program elements covered during test execution. The initial goal of SBFL techniques is therefore to identify program elements that are highly correlated with failing tests [119]. In order to determine the correlation between program elements and TC results, SBFL techniques utilize ranking metrics to pair a suspiciousness score with each program element, indicating how likely it is to be faulty. The rationale behind these metrics is that program elements frequently executed in failing TCs are more likely to be faulty. Thus, the suspiciousness score considers the frequency at which elements are executed in passing and failing TCs. Some of the more popular ranking metrics have been specifically created for the use in FL, such as Tarantula [78] and DStar [180], whereas others have been adapted from areas such as molecular biology, which is the case for Ochiai [130].

A study on developers' expectations on automated FL [94] shows that most of the studied developers view FL process as successful only if it:

- can localize faults in the Top-10 positions
- is able to process programs of size 100,000 LOC
- completes its processing in less than a minute

- provides rationales of why program elements are marked as potentially faulty

Considering these expectations, real-world evaluations [141] show that SBFL techniques are not yet applicable in practice. They are able to process large-size programs, but are not always able to locate the faults in top positions. This might be the consequence of considering the correlation, not causation. Although the goal of any FL technique is “to identify the code that caused the failure and not just any code that correlated with it” [118], SBFL techniques measure the correlation between program elements and test failures to compute suspiciousness scores. Thus, they do not control potential confounding bias [138]. Confounding bias is a distortion that modifies an association between an exposure (execution of a program element) and an outcome (program failure) because a factor is independently associated with the exposure and the outcome.

```
public void F1(int i) {
    if (i < 0) {
        ... // Faulty
        F2(x);
    } else {
        ...
        F3(x);
    }
}
```

Figure 1.1.: A Hypothetical Faulty Method with Two Branches

Using Figure 1.1, we exhibit an example of confounding bias in SBFL results. The code snippet indicates a hypothetical faulty program. Assume that a fault in method F1 propagates only through the left branch where method F2 is triggered, while the right branch where method F3 is called, executes correctly. Put differently, although F1 contains a fault, only those tests taking the left branch are failing. In this case, an SBFL technique gives the highest suspiciousness score to the method F2, since it’s executed more frequently in failing executions and less frequently in passing executions (F1: 1 failing, 1 passing, F2: 1 failing, and F3: 1 passing). However, method F1 is the faulty element.

In addition, for SBFL techniques, the granularity of the program elements in the program spectra is important, not only to the effectiveness of the system but also to the preferences of developers [93]. Kochhar et al. found that among surveyed developers, method, followed by statement and basic block were the most preferred granularities. But when it comes to the effectiveness of the system, the method and statement granularities may be too coarse- or fine-grained, respectively, to properly locate the faulty program elements [119, 135]. Unfortunately, there is no golden rule to say which granularity is the best for all contexts.

Despite ongoing research and improvements, the real world evaluations show that FL techniques are not always effective. Moreover, it is not clear why and under which circumstances they are effective. Thus, there is a need to raise this question and find influencing factors on the effectiveness of FL. Knowing these factors can help developers in

deciding whether to use FL techniques in their own domain or not.

1.2. Goal

The overarching goal of this thesis is to provide a framework for failure diagnosis which can be used as an assisting tool by developers to reduce failure analysis time, and therefore cost.

This framework will aid the decision making process of a debugging party regarding which data type and granularity to choose, and which clustering method and metric to employ for different domains. Moreover, it provides the opportunity for its users to predict the effectiveness of localization before adopting it. Also, it assigns a confidence factor to the results in each debugging session. Thus, users can better decide whether to use the framework or not and whether to consider the results or not.

1.3. Problem Statement and Research Questions

In this thesis, we tackle a problem, namely reducing failure analysis time, which was identified while conducting industry-oriented research. Together with the test engineers, we defined the problem statement:

Regardless of the domain, there are usually large numbers of tests that are frequently executed as regressions happen. In case of large numbers of failures, there are usually only a few underlying faults that cause a large number of tests to fail. Considering the significantly high execution time of the tests, it is not always possible to find the first fault, resolve it, and re-run all the tests.

Developers usually get quick and yet preliminary test results from the test runs, and use such information to resolve problems such as locating faults. Such a quick feedback approach may lower the development cost, but puts a lot of pressure on developers. In practice, the time resource allocated for each debugging session is usually limited and predetermined. Therefore, developers need an assisting tool to improve their productivity and reduce failure analysis time.

As described earlier, a lot of related work has been created in this direction (see Chapter 9). However, there are still some gaps that need to be addressed, summarized here:

- **Gap 1:** In case of large numbers of failing tests, there is a need for analyzing all the failing tests one by one since it is not clear how many of them should be inspected to find all the faults. Since large numbers of failures usually happen due to a few underlying faults, inspecting all the failing tests is redundant and time-consuming. Thus, there is a need for a technique that selects a proper representative subset of all failing tests which inspecting them could be enough to find and repair all the faults. This problem has been already addressed for black-box testing of database applications [153]. Nevertheless, there is a need for a different failure clustering

technique to apply in other domains where white box testing is being used. Moreover, there is no methodology to guide developers in applying failure clustering in their own domain.

- **Gap 2:** Existing failure clustering techniques work based on coverage data or domain-specific data [153]. There is a need for non-coverage data that is useful in failure clustering and is achievable when source code is not accessible.
- **Gap 3:** Existing granularity levels suggested for SBFL are either too fine- or coarse-grained [119]. There is a need for a new granularity that can cover more bug types.
- **Gap 4:** There is no way to predict whether SBFL is effective in a specific domain or on a specific program or not.
- **Gap 5:** There is no guideline for developers to help them in improving their code to facilitate fault localization.
- **Gap 6:** There are no indicators that shows how accurate the SBFL results are in each debugging session.

To address this problem statement, to bridge the gaps, and to achieve the goal of this thesis, the following research questions must be answered:

1. How effective are state-of-the-art techniques in reducing failure diagnosis time? (answered in Chapters 4, 6, 7, and 9)
2. Which clustering method and metric should one use to accurately group failing tests with respect to the faults that caused them? (answered in Chapter 4)
3. What kind of data can one use to accurately cluster failing tests with respect to the faults that caused them? (answered in Chapter 5)
4. How much reduction of failure diagnosis time is achievable using failure clustering? (answered in Chapters 4 and 5)
5. How to improve and make SBFL applicable in practice considering the above-mentioned developers' needs (regarding data availability, granularity level, trustworthiness (reliability), efficiency, and ability to provide rationale) [94]? (answered in Chapters 6, 7, 8, and 10)
6. Can we predict the effectiveness of SBFL? (answered in Chapter 8)
7. How can developers change their code to facilitate fault localization? (answered in Chapters 7 and 8)
8. How to use the results of failure clustering in fault localization? What are the benefits? (answered in Chapter 10)

9. Can we assign a confidence factor to the results of SBFL techniques? (answered in Chapter 10)

1.4. Solution

We introduce Aletheia, a framework for failure diagnosis. This framework would serve as an assisting tool for developers to empower them for fast and reliable failure diagnosis. Utilizing this framework, developers: (1) know which data (type and granularity) they need for further analysis, (2) are able to group failures based on their root causes, (3) are able to find more information about the root causes of each failing group. To this end, Aletheia has three main components: data generation, failure clustering, and fault localization.

Our framework is initially developed based on state-of-the-art failure diagnosis techniques. We implemented and evaluated diagnosis techniques for which we did not find a freely available implementation. The results led to an intuition that existing techniques might not satisfy developers' needs to apply them in practice.

To bridge the gap between theory and practice, *firstly*, we investigate the possibility of finding clustering method(s) that could be suitable for different domains and introduce a methodology for adapting this method(s) to different industrial domains with proper a priori parameter setting. Also, we add a representative selection mechanism to the clustering technique to eliminate the need for inspecting all the failing tests.

The failure clustering technique can be helpful in the following way. Assume 100 failing tests failed because of 6 distinct underlying faults. An ideal clustering finds 6 clusters from the failing tests, each cluster pointing to one fault. Selecting one representative test for each cluster means developers need to investigate only 6 failures rather than 100 failures to find all the faults.

Secondly, to improve the effectiveness of SBFL, we introduce a new granularity level, namely syntactic blocks. A syntactic block can be defined as a block of statements that syntactically belong together to form a program element, such as a method declaration, or a while loop. By considering different types of syntactic components in the program's source code, we are able to capture a wide range of faults, while providing a better insight into the possible location of the fault within the program element.

Besides, we augment SBFL with call and data-dependency graphs of failing tests to propose a re-ranking approach. This approach produces more accurate results augmented with the graphical representation of suspicious elements which can enhance users' understanding of the program's faulty behavior. Utilizing this technique, programmers find out the direction they should search for the fault. Moreover, we propose a confidence factor which demonstrates the probability of the fault being in the Top-10 ranks.

Finally and more importantly, we identify the influencing factors on the effectiveness of fault localization. We leverage these factors to predict the quality of fault localization. Thus, developers can decide whether to use the framework for fault localization or not. In addition, developers can use the model generated on these factors to facilitate fault

localization in their code.

The flowchart in Figure 1.2 depicts the process of using our failure diagnosis framework. All the needed inputs and provided outputs are shown in the flowchart. Based on different needs and conditions, users can utilize different components, failure clustering or fault localization, of the framework.

1.5. Contributions

The thesis makes the following contributions:

- **A failure clustering methodology.** To fill **gap 1**, with a methodological change w.r.t Jones' parallel debugging [77], we propose a failure clustering technique. We propose a methodology for adapting the idea of debugging in parallel to a real context, including an approach to choosing adequate parameter values and a tailored approach for measuring the quality of clustering. We augment it with adding a method for selecting representative tests in clusters as a final step to the clustering. Our approach is technically different from [153].
- **A collection of data sources for failure clustering.** To fill **gap 2**, we introduce a list of coverage and non-coverage data that are useful in the clustering of failing tests.
- **A new data granularity for failure diagnosis.** To fill **gap 3**, we propose a new granularity for program spectra called the syntactic block granularity which considers 18 different types of program elements.
- **A new ranking strategy.** We propose a ranking approach for SBFL techniques which leverages dynamic call and data-dependency graphs of failing executions.
- **A model to predict the effectiveness of SBFL.** To fill **gaps 4 and 5**, we propose a set of metrics which influence the effectiveness of SBFL. Using these metrics, we build a model to predict the effectiveness of SBFL. This model can be helpful in facilitating fault localization as well.
- **A framework for failure diagnosis.** We introduce a framework for failure diagnosis which puts failure clustering and fault localization in a pipeline.

Figure 1.3 highlights the contributions of this thesis in the failure clustering and fault localization areas. Parts of the contributions of this thesis have previously appeared in the following peer-reviewed publications, co-authored by the author of this thesis:

1. **M. Golagha**, A. Pretschner, D. Fisch, and R. Nagy, "Reducing Failure Analysis Time: an Industrial Evaluation," 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017.

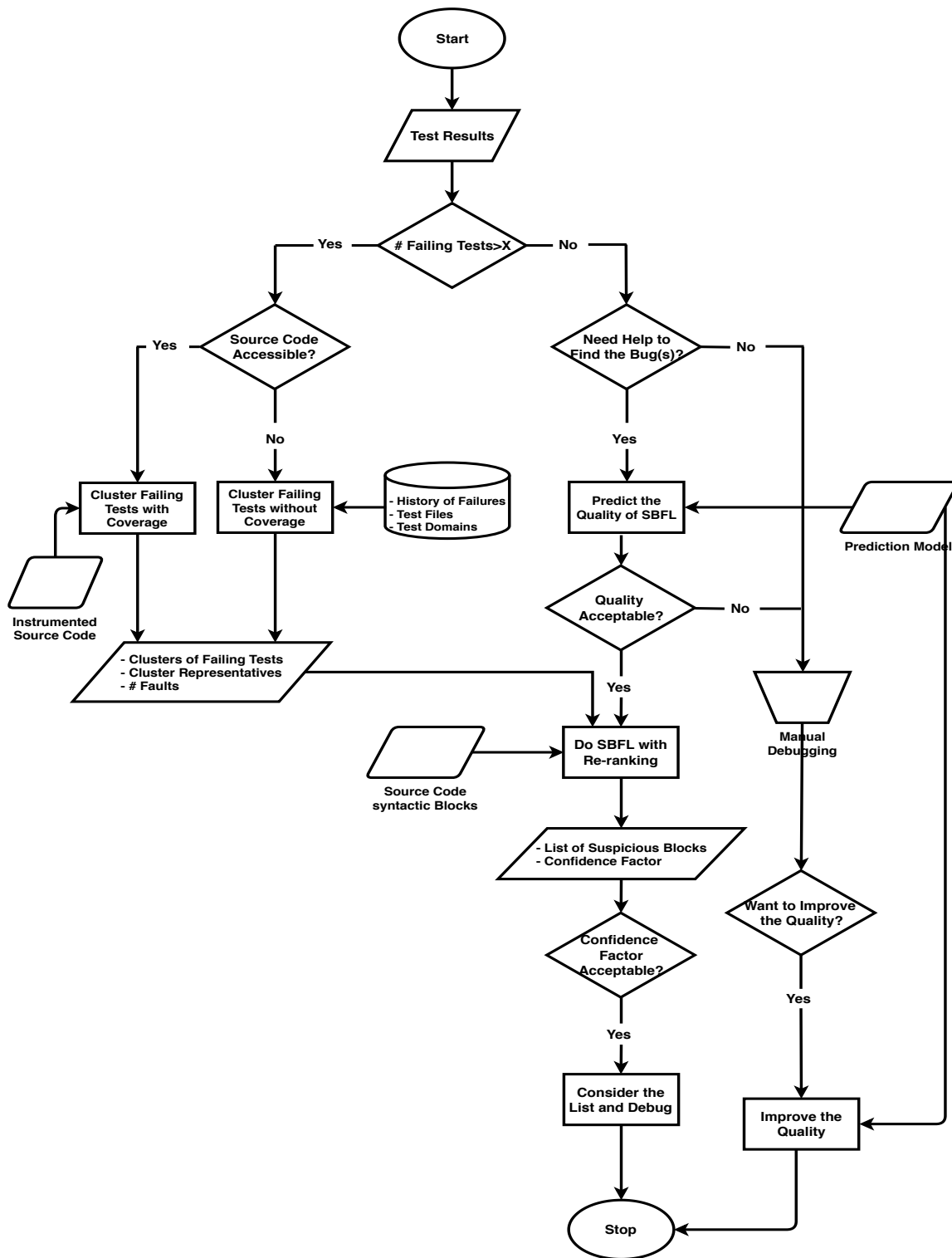


Figure 1.2.: The Framework's Flowchart - X is Predefined by Users.

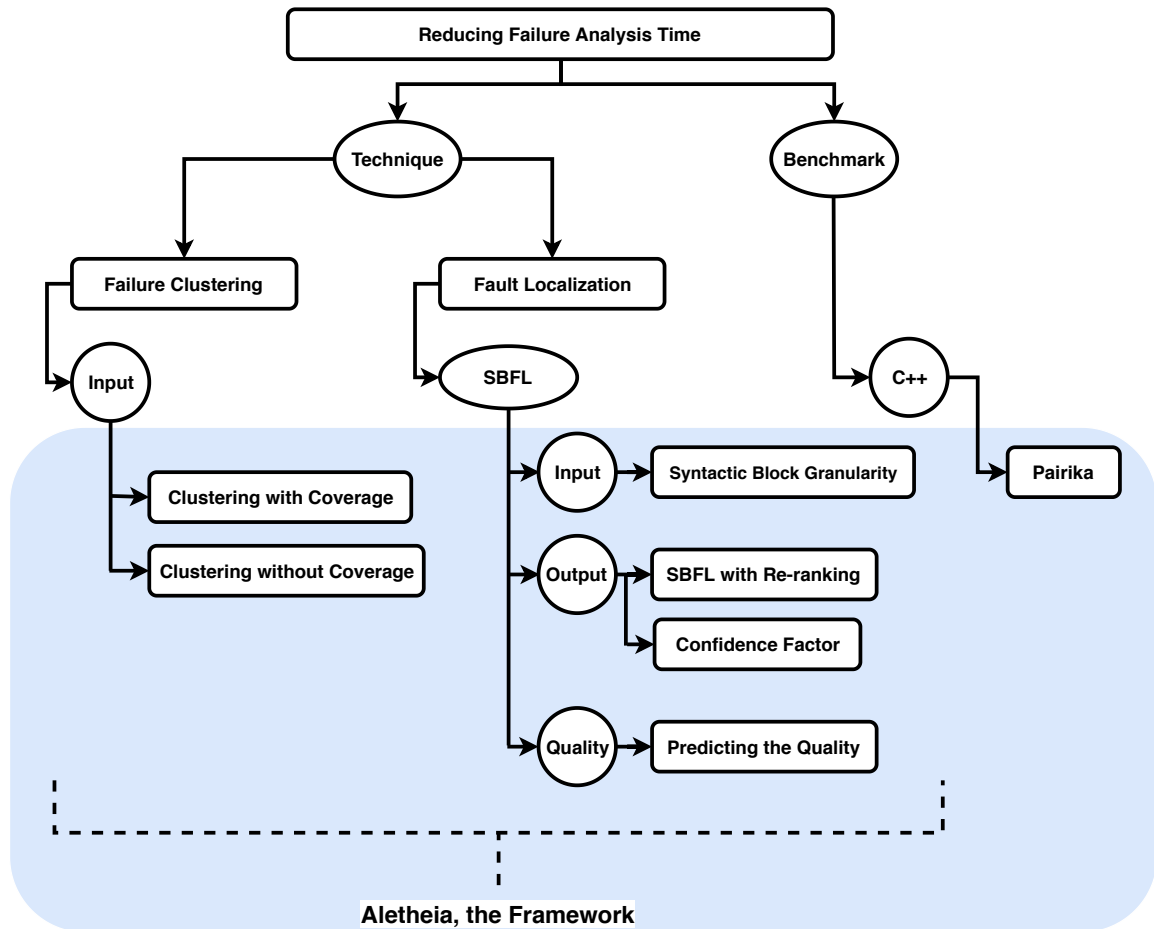


Figure 1.3.: Thesis Contributions

2. **M. Golagha**, "A Framework for Failure Diagnosis," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017.
3. **M. Golagha** and A. Pretschner, "Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective," 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017.
4. **M. Golagha**, A. M. Raisuddin, L. Mittag, D. Hellhake, and A. Pretschner, "Aletheia: A Failure Diagnosis Toolchain," 2018 ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE), 2018.
5. Md. R. Rahman, **M. Golagha**, and A. Pretschner, "Pairika: A Failure Diagnosis Benchmark for C++ Programs," 2018 ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE), 2018.

6. **M. Golagha**, C. Lehnhoff, A. Pretschner, and H. Ilmberger, “Failure Clustering Without Coverage”, 2019 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2019.
7. **M. Golagha**, A. Pretschner, and L. C. Briand, “Can We Predict the Quality of Spectrum-based Fault Localization?”, 2020 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2020.

1.6. Summary of Results

Our evaluation results show that utilizing our failure clustering approach, we are able to reduce more than 80% (using coverage data) or 60 % (using non-coverage data) of the analysis time.

Our re-ranking strategy which leverages dynamic call and data-dependency graphs of failing executions outperforms the traditional SBFL techniques and also causal inference-based techniques. Our experiments show that to localize a bug, a programmer should first look for the most suspicious method on the call graph then look upward and inspect its parent and grandparents instead of inspecting the ranking list in a linear fashion.

We found that the syntactic block granularity exhibits best-case absolute ranking behavior similar to the method granularity while having a wasted effort (estimated effort for a developer to find the origin of a fault) equivalent to, if not better than, the statement granularity. Furthermore, it covers more types of faults than both existing granularities. Finally, when compared to the method granularity, it exhibits up to a 92.48% improvement when it comes to the locality of the program elements to the fault, a characteristic that provides the user with a better insight into the possible location of the faults.

Our analysis on the effectiveness of SBFL show that a combination of 4 Static, 4 Dynamic, and 2 Test metrics, gives us a model with excellent discrimination power (AUC=0.86) which can be used in 2 ways: as a prediction model for the effectiveness of fault localization, and as a confidence factor for the results of fault localization. These 10 metrics are the most influential metrics on the effectiveness of fault localization. Thus, they can be used as a guide to improve the code quality for more effective fault localization.

1.7. Structure

Chapter 2 provides an overview of existing failure diagnosis techniques and their shortcomings. Chapter 3 introduces the benchmarks used in the evaluations. Chapters 4 and 5 describe failure clustering techniques using coverage and non-coverage data respectively. Chapter 6 describes the new granularity level to improve SBFL. Chapter 7 describes a new ranking strategy to improve SBFL results. Chapter 8 introduces a model to predict the quality of fault localization. Chapter 9 presents related work. Chapter 10 presents conclusions, insights and future work.

2. Background

This chapter presents an introduction to the failure clustering and fault localization techniques and their advantages in failure diagnosis. Parts of this chapter have previously appeared in publications [55, 58, 54, 56, 150], co-authored by the author of this thesis.

2.1. Failure Clustering Background

As discussed in Chapter 1, Jones et al. [77] introduced two clustering approaches. Using the first technique, they cluster failures based on execution profiles and fault localization results. Their second technique suggests to only use the results of fault localization. The further evaluations [68, 58] showed that the first technique is effective if it is adapted to the context and if it is used for clustering failures rather than debugging and localizing faults. Therefore, in our thesis, we utilize Jones' first clustering approach as the base of our work and build on it (see Chapters 4 and 5).

2.1.1. Agglomerative Hierarchical Clustering

Based on Jones et al. work, we use the Hierarchical Clustering (HC) algorithm to cluster failing tests. HC is an unsupervised machine learning [116] algorithm which measures the distance between data objects to find their similarity. The goal of HC is to compute a *dendrogram*, a tree structure of data objects. The number of clusters can vary from one to the number of individual data objects (m).

Agglomerative clustering is a 'bottom-up' Hierarchical Clustering approach. The algorithm starts with m clusters that each includes only one data object. Iteratively, clusters with the smallest inter-cluster distance are merged until only one cluster remains.

In our work, each data object is a failing test. Thus, each single failure at first is in its own cluster. Using some distance metrics, the similarity between clusters is measured, and the two most similar clusters are merged to build a new cluster. Figure 2.1 shows a cluster tree for hypothetical failing tests T2, T4, T5, and T6. In the first iteration, the T5 and T6 are merged and formed cluster C3, then T2 and T4 are merged into cluster C2. In the third iteration, C2 and C3 are merged forming a new cluster C1.

Hierarchical clustering does not need a predefined number of clusters, k . However, since we want a partition of (by definition: disjoint) clusters, the hierarchy needs to be cut at

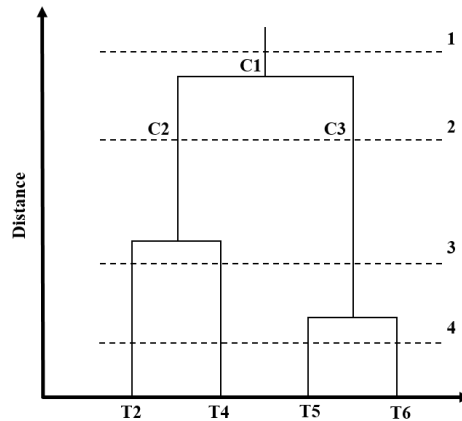


Figure 2.1.: Hierarchical Clustering of 4 Failing Tests

some point (if there is more than one underlying fault). There are a number of criteria in the literature to determine the cutting point [116]. Cutting the tree at a given height will partition the tree at a selected precision. For instance, in Figure 2.1 example, cutting after the first row ($n=2$) of the tree will yield two clusters, C2 and C3, and cutting after the second row ($n=4$) will yield four clusters, T2, T4, T5, and T6. The second cutting distance leads to more fine-grained clusters.

Clustering Methods

Clustering method defines how given a distance metric, the distance between clusters is calculated. In our experiments, we used the following methods [116] (considering clusters C2 and C3 in Figure 2.1 as an example):

- **Single:** The distance between two clusters C2 and C3 is defined as the minimum distance between any two objects of C2 and C3.
- **Complete:** The distance between C2 and C3 clusters is defined as the maximum distance between any two objects of these clusters.
- **Average:** The distance between C2 and C3 clusters is defined as the average distance between objects of these two clusters.
- **Weighted:** The distance between two clusters C2 and C3 is defined as the mean of the distances between T2, T4, T5, and T6, since cluster C2 was formed by T2 and T4 and C3 was formed by T5 and T6 in the previous iterations.
- **Centroid:** The distance between C2 and C3 clusters is defined as the distance between the two centroids of these clusters.

Distance Metrics

Distance metric defines how to calculate the distance between any two objects represented as numerical feature vectors [116]. The distance between two vectors u and v is defined as:

- **Euclidean:**

$$\sqrt{\sum_{i=1}^n (u_i - v_i)^2} \quad (2.1)$$

- **Squared Euclidean:**

$$\sum_{i=1}^n (u_i - v_i)^2 \quad (2.2)$$

- **Cityblock:**

$$\sum_{i=1}^n |u_i - v_i| \quad (2.3)$$

- **Cosine:**

$$\frac{\sum_{i=1}^n u_i \cdot v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}} \quad (2.4)$$

- **Correlation:**

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{\sqrt{\sum_{i=1}^n (u_i - \bar{u})^2} \sqrt{\sum_{i=1}^n (v_i - \bar{v})^2}}, \quad (2.5)$$

where \bar{u} represents the mean value of the elements of u .

- **Hamming:**

$$\sum_{i=1}^n [u_i \neq v_i] \quad (2.6)$$

- **Jaccard:**

$$\frac{U \cap V}{U \cup V} \quad (2.7)$$

- **Chebyshev:**

$$\max_i (|u_i - v_i|) \quad (2.8)$$

- **Canberra:**

$$\sum_{i=1}^n \frac{|u_i - v_i|}{|u_i| + |v_i|} \quad (2.9)$$

- **Braycurtis:**

$$\frac{\sum_{i=1}^n |u_i - v_i|}{\sum_{i=1}^n |u_i + v_i|} \quad (2.10)$$

- **Yule:**

$$\frac{2 \times c_{TF} \times c_{FT}}{n}, \quad (2.11)$$

where c_{TF} is the sum of occurrences where $x[i] == True$ and $y[i] == False$ for two binary feature vectors x and y of length i .

2.1.2. Evaluation Metrics

To evaluate the performance of the clustering, we considered two groups of metrics, one group to measure the effectiveness from the scientific point of view and one group to measure the effectiveness from the practical point of view.

Practice Oriented Metrics

From the perspective of a tester, three questions should be answered:

1. How many of the existing faults do we find analyzing only the cluster representatives?
2. How many tests are assigned to the correct faults if we analyze the representative failures and assign the found fault to all of the cluster members?
3. How much reduction is achievable using the clustering tool?

To answer these questions, we measure the following metrics respectively:

FoundCauses: shows the ratio of the faults found analyzing only the representatives.

$$FoundCauses = \frac{|F_{found}|}{|F_{total}|}, \quad (2.12)$$

where F_{found} is the set of unique faults found through analyzing the representative tests and F_{total} is the set of unique underlying faults that are found manually by developers and stored in the database. Score "1" means all the faults have been found.

Purity: Purity shows how many failures are assigned to the correct faults if analyzing only the representative tests and assigning the same fault to all the cluster members. To compute Purity, each cluster is assigned to the class which is most frequent in the cluster [117]. A class is a fault in our case. The accuracy of this assignment is measured by counting the number of correctly assigned objects dividing by the total number of failures.

$$Purity(\Omega, \mathbf{C}) = \frac{1}{N} \sum_k \max_j |\omega_k \cap c_j|, \quad (2.13)$$

where N is the total number of failures, $\Omega = \{\omega_1, \omega_2, \dots, \omega_k\}$ is the set of K clusters and $C = \{c_1, c_2, \dots, c_j\}$ is the set of J classes. Purity score "1" means all the members in the cluster failed because of the same reason and all of them have been assigned to the same fault.

Achieved Reduction (ARed): We introduce a Reduction (Red) metric which measures the avoided effort as complement of the ratio of the number of clusters to the number of test cases:

$$Red = \left(1 - \frac{K}{\# \text{ of } TCS}\right) \times 100. \quad (2.14)$$

Note that Red alone is not sufficient for judging the effectiveness. To realize how successful we are in the reduction of analysis time, it is also necessary to know the ideal reduction in each case. As examples, first consider a test suite with 8 failing tests and 6 faults. In an ideal clustering, there should be 6 clusters each one pointing to 1 fault which means investigating 6 tests rather than 8 tests. Thus, the maximum possible reduction is 25%. As a second example, consider a test suite with 20 failing tests and 2 faults. In this case, the maximum possible reduction with an ideal clustering is 90%. Just considering the absolute effective reduction Red does not tell us *how good we could have been*.

Therefore, we measure how much of the ideal reduction ($IRed$) has been achieved, and name it Achieved Reduction ($ARed$):

$$ARed = \frac{Red}{IRed} \times 100 \quad (2.15)$$

where $IRed$ is:

$$IRed = \left(1 - \frac{J}{\# \text{ of } TCS}\right) \times 100 \quad (2.16)$$

In addition, since more clusters mean more representative tests and thus less reduction in analysis time, $ARed$ can also serve as an easily interpretable measure that shows how much finding extra clusters affects the effectiveness of our approach.

We, finally, combine these three evaluation metrics to a single metric. A single performance metric facilitates the evaluations of different parameter combinations when fitting the clustering model. Thus, we define the **Performance** metric as:

$$Performance = w_{fc} \times FoundCauses + w_p \times Purity + w_{ar} \times ARed, \quad (2.17)$$

where w_{fc} , w_{ar} , and w_p are the weights that can be defined by the developers based on their needs. After discussion with industry experts and training with different values, we found the best combination as $w_{fc} = 0.4$, $w_{ar} = 0.4$, and $w_p = 0.2$. An important fact to consider when choosing the performance weights is that both the FoundCauses and the Purity metrics (unlike ARed) favor the larger number of clusters and a few members in each cluster. For instance, having each failing test in a separate cluster leads to *Purity* score "1" and subsequently FoundCauses score "1" but ARed score "0".

Science Oriented Metrics

Typical objectives in clustering are high intra-cluster similarity and low inter-cluster similarity. However, good scores on these criteria do not necessarily mean good effectiveness in our application. To do a proper evaluation, we need to consider both scientific and industrial objectives.

We chose *purity*, *F-measure* and *entropy* from the literature as our science oriented evaluation metrics. To compute purity, each cluster is assigned to the class which is most frequent in the cluster. A class is an underlying fault in our case. Then, the accuracy of this assignment is measured by counting the number of correctly assigned objects dividing by N [117] where N is the number of objects. Let $\Omega = \{\omega_1, \omega_2, \dots, \omega_K\}$ be the set of K clusters and $\mathbf{C} = \{c_1, c_2, \dots, c_J\}$ the set of J classes. Then purity is:

$$Purity(\Omega, \mathbf{C}) = \frac{1}{N} \sum_k \max_j |\omega_k \cap c_j| \quad (2.18)$$

If the number of clusters is large, achieving high purity is easier. If each object is in its own cluster, purity is 100%. Purity is the most important factor from a practical point of view since it means all the failures in one cluster failed because of the same reason. However, there is a trade-off between the quality of clustering and the number of clusters. Since fewer clusters mean more reduction in analysis time, which is our objective, we need a measure for this trade-off.

In the clustering literature, the F-measure [117] is proposed to this end:

$$\mathbf{F}_1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (2.19)$$

where $precision = TP/(TP + FP)$ and $recall = TP/(TP + FN)$.

In our case, a true positive (TP) means assigning two failures grounded in the same fault to the same cluster; a true negative (TN) means assigning two failures grounded in different faults to different clusters; a false positive (FP) means assigning two failures grounded in different faults to the same cluster; and a false negative (FN) means assigning two failures grounded in the same fault to different clusters. We also use the entropy metric proposed by Rogstad and Briand [153] to capture the trade-off: For each of the K clusters in Ω , they compute the number of failures of type i (failing because of underlying fault i) that belongs to cluster c_j (f_{ij}) divided by the total number of failures of type i ($|f_i|$):

$$E_F(f_i, C) = - \sum_{j=1}^K \left(\frac{f_{ij}}{|f_i|} \right) \log \left(\frac{f_{ij}}{|f_i|} \right), \quad (2.20)$$

and total deviation entropy is:

$$E_{F-TOT}(F, C) = - \sum_{i=1}^J E_F(f_i, C). \quad (2.21)$$

We provide these values chiefly for comparison purposes with existing work.

2.2. Fault Localization Background

One of the most popular subsets of techniques for automated fault localization (FL) is spectrum-based fault localization, known as SBFL [182].

2.2.1. Spectrum-based Fault localization

SBFL techniques analyze the hit spectrum of a program to find the fault. By measuring the similarity between the spectrum and the verdict vector, one can identify program elements which correlate with test failures. An example of a hit spectrum is shown in Table 2.1. Each \bullet in the table means that the respective element “e” (e. g., statement, method, basic block etc) was hit in the respective test run “t”.

Table 2.1.: A Hypothetical Hit Spectrum

Element	Test Cases					
	t1	t2	t3	t4	t5	t6
e1	•	•	•	•	•	•
e2		•	•	•	•	
e3		•	•	•	•	
e4	•	•			•	•
e5		•				
Verdict	F	P	P	P	P	F

There are several metrics in the literature to measure the similarity[182]. DStar, Ochiai, and Tarantula are three of the most popular and best-performing metrics in recent studies [141].

$$DStar = \frac{(N_{CF})^*}{N_{UF} + N_{CS}},$$

$$Tarantula = \frac{\frac{N_{CF}}{N_f}}{\frac{N_{CF}}{N_f} + \frac{N_{CS}}{N_s}},$$

$$Ochiai = \frac{N_{CF}}{\sqrt{N_f * (N_{CF} + N_{CS})}},$$

where N_f is the number of failing tests, N_s is the number of passing tests, N_{CF} is the number of failing tests that cover the element, N_{CS} is the number of passing tests that cover the element, N_{UF} is the number of failing tests that do not cover the element. DStar metric takes a parameter *. The nominator is then taken to the power of *. There is no significant difference between these metrics [141]. Table 2.2 shows the suspiciousness scores and ranks of program elements in Table 2.1 using Ochiai metric. As the ranks indicate, element e4 is the most suspicious element.

Table 2.2.: Ochiai Suspiciousness Scores

Element	Suspiciousness	
	Score	Rank
e1	0.577	2
e2	0	3
e3	0	3
e4	0.707	1
e5	0	3

2.2.2. Method-Level Fault Localization with Causal Inference

Causal inference FL techniques intend to improve SBFL effectiveness by considering the call and data-dependency graphs of test runs. This subset of techniques considers the causal relationship between method coverage and test results [118]. The first causal-based technique was proposed by Baah et al. in [7]. Later, Shu et al. extended it to a method-level approach [162] named MFL. We describe MFL briefly in the following.

MFL starts with constructing a causal graph. A causal graph models the dependencies between random variables. Nodes denote random variables. A directed edge between node X and Y means that X may cause Y . For an FL problem, the causal graph is constructed by combining the dynamic call and data-dependency graphs of a program (P). In MFL, each node is representing a method (M) triggered in a test run. Every call from one method to another is reflected by adding an edge from caller to callee. Also, for any explicit data-dependency between two methods, one edge is added. Finally, an outcome node (Y), and an edge from each method to the outcome is added to the graph. Y is equal to 0 if the test is failing, and 1 if it is passing. In the next step, utilizing Pearl’s Backdoor Criterion [139] the potential confounders are listed for each method.

The final step is estimating a causation-based suspiciousness score for all the methods covered in at least one failing TC. Put it differently, the aim is to obtain a causal-effect estimate of M on the outcome of P that is not subject to severe confounding bias. To this end, MFL uses the following regression model for an experiment with S tests:

$$Y = \alpha + \tau_M T_M + \beta_M X_M + \epsilon,$$

where T_M is the treatment variable which is 1 when the method is executed in the test and 0 otherwise, X_M is a $S \times 1$ vector with an entry for each method on the confounders list (1 if the confounding method is executed in the test, 0 otherwise). Linear regression is then used to estimate other variables. τ_M describes how significant the influence of the treatment variable T_M is on the test failure ($Y = 0$). Therefore, it is used as suspiciousness score for method M . The model is fit separately for each method.

We implemented MFL based on the pseudocode given in [162]. When applying it, we noticed that the confounder selection part was costly. Thus, it was not always possible to calculate the suspiciousness for all the methods in a failing test. To mitigate this issue, instead

of linear regression, we utilized Random-Forest-Regressor to estimate the failure-causing effect of M (τ_M). We used the implementation of Random-Forest-Regressor provided by the Akelleh Causality tool ¹. This estimator uses the average value of training multiple decision trees. The training data is extracted randomly from the whole data set.

Furthermore, we changed the expensive confounder selection part. We improved it by parallelizing computations of the weak-positivity table and calling it only on-demand. These changes improved the performance and made it possible to run this approach on large projects. We call the improved version Improved-MFL in the rest of this thesis.

2.2.3. Evaluation Metrics

When evaluating our contributions in improving SBFL, we make the simplifying assumption of perfect bug detection, in that, if given a program element contains a fault, the user will be able to localize the fault 100% of the time [182]. This allows us to evaluate the effectiveness of each FL technique, without having to worry about the complexity of the faults themselves.

Furthermore, due to the fact that many of the faults in practice spread across multiple lines, there will often be multiple program elements that localize the fault. To deal with this, all of our metrics are relative to the most suspicious (highest scoring), fault-localizing element, which we will denote as e^* .

When evaluating the results, we consider the following metrics extracted from [182, 135, 142, 115].

EXAM Scores

EXAM score or wasted effort is defined by Wong and Debroy as “the percentage of executable statements that have to be examined until the first statement containing the bug is reached” [179]. Current studies assume that developers have a perfect understanding of fault and can recognize the fault as quickly as they find the location of bug while going through the ranking list. With this premise, developers have to inspect all the elements that are ranked higher than the faulty element. The number of inspected non-faulty elements is defined as the wasted effort for the developer [85]. Current research commonly puts the wasted effort in relation to the total number of ranked elements. In the case of ties, a distinction has to be made. Therefore we use three variants called O-EXAM, P-EXAM, and Δ -EXAM [115].

O-EXAM, or Optimistic EXAM score, takes a best-case approach to tie breaking, where it is assumed that the faulty program element e^* is chosen first from the list of tied elements. Therefore, it only considers the wasted effort from all elements with a suspiciousness score strictly higher than the fault containing element e^* .

$$O-EXAM = \frac{\text{best-case wasted effort}}{\text{total elements}} \times 100 \quad (2.22)$$

¹<https://github.com/akelleh/causality>

P-EXAM, or Pessimistic EXAM score, takes a worst-case approach to tie breaking, where it is assumed that the program element e^* is the last element chosen from the list of tied elements. It considers the wasted effort from the optimistic version, plus the wasted effort from all other tied program elements. However, in the case of a tie containing multiple faulty elements, the wasted effort from all other non-faulty elements is instead calculated.

$$P\text{-EXAM} = \frac{\text{worst-case wasted effort}}{\text{total elements}} \times 100 \quad (2.23)$$

Δ -EXAM is defined as follows:

$$\Delta\text{-EXAM} = P\text{-EXAM} - O\text{-EXAM} \quad (2.24)$$

For all three EXAM scores, we provide the average and median values.

Best and Worst Absolute Rank

To account for interest drop-off in programmers' investigation and support scalability to larger programs, Parnin et al. suggest the usage of absolute rank [135]. Absolute rank considers the position of the program element e^* in the ranked list of suspicious program elements.

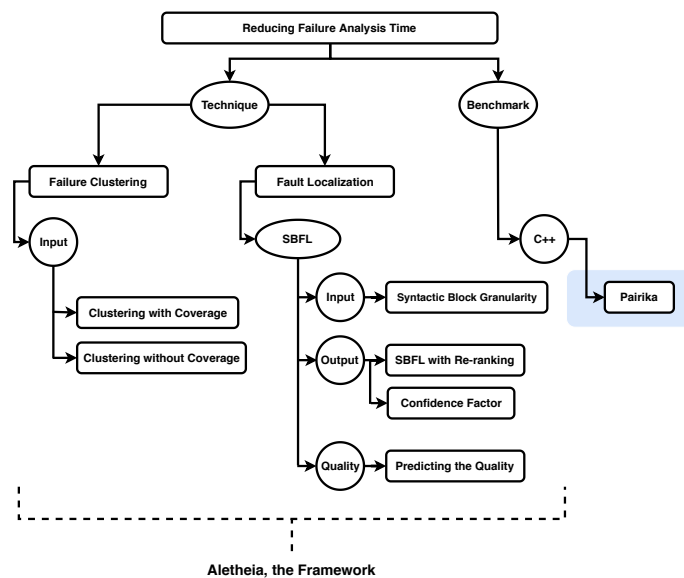
However, like the EXAM scores, we need a distinction in the case of ties. Therefore, to get a better understanding of the best- and worst-case effectiveness of the evaluated techniques, we calculate the average and median of the **best absolute rank** and, **worst absolute rank**. Best absolute rank assumes that the faulty element e^* is the first to be examined among the elements of the same suspiciousness, whereas worst absolute rank, like P-EXAM score, assumes the faulty element e^* is examined after all other tied, non-faulty elements.

Top-N Counts

To get a better idea of the best and worst absolute ranks, we count the number of bugs whose absolute rank of the program element e^* is in the top N ranks. More specifically, we consider **Top-1**, **Top-5**, and **Top-10** counts for both the best and worst absolute rank.

3. Case Studies

This chapter presents an overview of the case studies used in the evaluations. Parts of this chapter have previously appeared in peer-reviewed publications [150, 54], co-authored by the author of this thesis.



In this chapter, we describe the case studies and benchmarks we utilized to evaluate our solution ideas. Each chapter’s solution idea is evaluated using only one of the following case studies. However, we used a combination of these case studies to evaluate our framework in Chapter 10. All of these case studies are real.

3.1. Large Car Manufacturer (LCM)

Testing and debugging automotive Cyber Physical Systems (CPS) are becoming more challenging due to rapid innovation. These innovations are applied with the aid of applications and software features added to cars. These applications execute on dozens of programmable electronic control units (ECU) that communicate through various communication buses such as CAN, FlexRay, LIN and automotive Ethernet. To test ECUs, developers perform

multiple levels of testing, ranging from unit tests to acceptance tests. In practice, there are always limited time and resources available to analyze failures and find underlying faults.

SiL (software-in-the-loop) and HiL (hardware-in-the-loop) are two levels of testing that encompass huge numbers of tests with high execution times. Testing an ECU on the SiL level means that its *software components* are tested within a simulated environment model but without any actual hardware [17]. HiL testing aims at testing the integration of hardware and software of an ECU in a simulated environment. At the HiL level, the software hence runs on the final ECU, again within a simulated environment [17]. TCs in these two levels are frequently re-run as regression tests. Due to large number of TCs at the SiL and HiL levels, their execution time is significant. Therefore, in the process of analyzing failures, it would be too expensive to execute tests immediately after correcting a single fault. In practice, to remove all the underlying faults, testers dig into failing tests one by one to make sure they have found and resolved all the faults before re-running the tests. This process takes a significant amount of time to be finished.

In the following, we review the basics in automotive software test and integration.

Body Domain Controller

Usually, each ECU is responsible for regulating a separate sub-set of car features. To enable ECUs to exchange data about their states, multiple communication buses (e.g. CAN, FlexRay, Automotive Ethernet) are used. The communication buses create a *boardnet* together with the ECUs.

In the boardnet of each car, the Body Domain Controller (BDC) is one of the fundamental ECUs that links all major buses to allow board-wide interactions and regulates the *body* features that define several fundamental functions that each car provides. Each body function is part of a *component* (e.g. air conditioning, seat features, sunblinds). BDC consists of 7 domains and over 100 components.

Automated Software Integration and Analysis

To ensure quality and safety, ECU software should be tested in several stages during its development. Following a continuous integration approach, automotive companies need to continuously select, compile, and execute a large number of TCs. Tests are either executed in a SiL or HiL environment.

TCF - Test Case Framework

LCM uses a black-box testing method to verify the correctness of ECU software during its development. Boardnet messages are used for this purpose to stimulate the ECU from the external environment. Then, responses of the ECUs are compared to the expected values. An abstract domain-specific language called TCF (Test Case Framework) facilitates coding and running TCs for the BDC [124]. In TCF, *Actors* encapsulate communications with ECUs.

As shown in Figure 3.1, each Actor offers a number of actions to regulate a particular ECU function, e.g. to alter the audio module volume. In the HiL level of testing, these actions are converted into bus messages using a *Mappings* tool. A mapping scenario for Figure 3.2 is shown in Figure 3.1. Mapping is used to translate the behavior of an Actor to an I/O signal by specifying the type of bus (e.g. 'LIN'), the particular bus (e.g. 'KLIN8'), the messages (e.g. '0x2A'), and the name of the bus.

```
Audio_Volume {
  VolumeUp(steps)
  VolumeDown(steps)
  VolumeNoDirection(steps)
  NoAction
  Invalid
}
```

Figure 3.1.: TCF Actor

```
HiLMappings {
  Set Audio_Volume.VolumeDown(steps) {
    LIN.KLIN8.0x2A.ST_DIRRT_AUDCU_LIN = steps
  }
  Check Audio_Volume.VolumeDown(steps) {
    LIN.KLIN8.0x2A.ST_DIRRT_AUDCU_LIN = steps
  }
}
```

Figure 3.2.: TCF HiL Mapping

The other two important concepts used in the TCF language are *Codings* and *Alternatives*, as shown in Figure 3.3. Codings are needed to define which peripheral hardware is connected (e.g., the audio module is connected and controlled via LIN bus). *Alternatives* enable developers to define different situations for TC execution (e.g. car condition 'parking' vs. 'driving'). For each Alternative, a distinct test is generated at compile time. For instance, the 'Audio_Example.tcf' file, illustrated in Figure 3.3, leads to the generation of two tests: 'Audio_Example.a1' and 'Audio_Example.a2'.

Data sets

In this thesis, we evaluate our ideas in reducing failure analysis time for both BDC SiL and HiL regression testing and debugging.

To generate the study data sets, we used the current as well as some old versions of the code and reported bugs available in the repositories of LCM. Table 3.1 shows information of the software components under test along with their SiL tests. We have changed the names of the components for confidentiality reasons. We performed our approaches in parallel

3. Case Studies

```
TestCase Audio_Example {
  HiLCoding {
    BE_AUDIO_VERB = 01 // connecting audio module
    BE_AUDIO_LIN_VAR = 01 // connecting LIN bus
  }
  Execute {
    Alternatives {
      PWF = ST_CON_VEH_PARK_IO // parking
      PWF = ST_CON_VEH_DRIV // driving
    }
    Audio_Volume = VolumeDown(15)
    Run($ShortMainTaskCycle)
    Audio_Volume == VolumeDown(15)
  }
}
```

Figure 3.3.: TCF File

Table 3.1.: Software Components of the Case Study

	Lines of Code	Number of TCs
SWC1-OV1	≈ 145,952	1103
SWC1-OV2	≈ 145,952	1103
SWC1*	145,952	1103
SWC2*	113,470	1303
SWC3	14,434	23
SWC4	59,349	890
SWC5	42,308	793
SWC6	87,984	687
SWC7	7,164	424
SWC8	29,930	234
SWC9	59,349	163

Table 3.2.: SiL Builds

Build	Related Component	# of Failing Tests	# of Faults
1	SWC1-OV1	24	2
2	SWC1-OV2	240	3
3	SWC1	32	8
4	SWC2	25	4
5	SWC3	7	1
6	SWC4	39	5
7	SWC5	30	3
8	SWC6	19	4
9	SWC7	66	1
10	SWC8	9	4
11	SWC9	8	1

with the partner's current process of failure analysis. This allows us to use the developers' analysis reports as ground truth.

All in all, we had access to 11 SiL and 86 HiL builds (test runs). Considering SiL builds, we had access to 499 failing tests and 46 faults (see Table 3.2). Considering HiL builds, we had access to 8743 failing tests and 1531 faults. Due to large amount of HiL data, we reserved 10% of the data for test purposes and used the remaining 90% to train the algorithms. As the result, the training set consists of 77 builds, more than 15000 passing tests, 7837 failing tests and 1360 faults as shown in Table 3.3.

Table 3.3.: HiL Builds [102]

Build	# of Failing Tests	# of Faults	Phase	Build	# of Failing Tests	# of Faults	Phase
3	131	17	Train	113	60	27	Train
10	38	9	Train	114	37	20	Test
12	33	7	Train	115	41	18	Train
18	38	10	Train	116	147	13	Train
21	55	11	Train	117	9	2	Train
24	159	15	Train	118	261	10	Train
26	167	15	Train	119	195	17	Train
29	127	16	Train	121	178	16	Train
31	142	9	Test	122	107	17	Train
34	53	19	Train	123	125	12	Train
37	50	6	Train	124	185	10	Test
39	74	9	Train	125	161	29	Train
42	43	24	Train	127	206	27	Train
46	101	15	Train	128	120	22	Train
47	8	7	Train	129	99	17	Train
48	69	15	Train	130	149	21	Train
53	42	6	Train	131	164	22	Train
54	77	27	Test	132	118	32	Train
55	58	17	Train	133	222	18	Train
56	55	12	Train	134	84	23	Test
63	103	13	Train	135	71	24	Train
68	56	15	Train	136	117	18	Train
69	48	8	Train	138	66	19	Train
70	77	20	Train	139	107	27	Train
73	21	9	Train	140	90	13	Train
74	30	15	Train	141	57	24	Train
75	86	12	Test	142	90	37	Train
84	99	1	Train	143	52	25	Train
85	171	26	Train	144	121	33	Test
89	181	23	Train	145	63	16	Train
91	38	8	Train	146	89	26	Train
92	173	29	Train	147	89	20	Train
95	95	29	Train	149	166	22	Train
97	79	22	Train	150	90	18	Train
99	119	25	Train	151	183	14	Train
102	96	25	Test	152	191	26	Train

Continued on next page

Table 3.3 Continued from previous page

Build	# of Failing Tests	# of Faults	Phase	Build	# of Failing Tests	# of Faults	Phase
103	82	25	Train	153	90	10	Train
107	164	35	Train	154	78	13	Test
108	83	30	Train	155	59	8	Train
109	59	30	Train	156	118	27	Train
110	101	17	Train	158	70	17	Train
111	56	26	Train	162	264	5	Train
112	7	2	Train	163	210	5	Train

3.2. Defects4J

Defects4J [82] is the most popular benchmark of real Java bugs in recent studies to evaluate new ideas in failure diagnosis. Table 3.4 summarizes some characteristics of Defects4J's projects.

Table 3.4.: Defects4J Projects [82] [31]

Project	Code LoC	Test LoC	# of Test	# of Faults
Closure	90K	83K	7,927	133
Lang	22K	6K	2,245	65
Math	85K	19K	3,602	106
Time	28K	53K	4,130	27
Chart	96K	50K	2,205	26
Mockito	12K	11K	1,854	38

Defects4J consists of 395 real faults from 6 open source Java projects: JFreeChart (26 bugs), Closure Compiler (133 bugs), Apache Commons Lang (65 bugs), Apache Commons Math (106 bugs), Mockito Testing Framework (38 bugs), and Joda Time (27 bugs). In each buggy version, there is exactly one bug per program under consideration. For each bug, Defects4J provides the buggy program version and its corresponding fixed version. Furthermore, Defects4J bugs are 1) related to source code (i.e., fixes within configuration files, documentation, or tests are not included), 2) reproducible (each bug contains at least one test that exposes the bug), and 3) isolated (patches do not include unrelated changes to the bugs such as refactoring or feature additions). In this thesis, when referring to different buggy versions, we use the notation of [project name]-[bug id], for instance, Chart-22 or Lang-27.

We chose Defects4J due to it being the largest available database of real bugs for Java, as well as the fact that it has been used in many other recent studies [195, 142, 165, 6, 95, 145].

3.3. Pairika

Pairika is the first publicly accessible benchmark for C++ programs. It contains 40 bugs extracted from 7 modules of OpenCV project with more than 490 KLoC and 11129 tests. Each bug is accompanied by at least one failing test.

OpenCV is a library of programming functions aimed at real-time computer vision written in C and C++. The library runs under Linux, Windows and Mac OS X. It has 52 modules [131]. Table 3.5 demonstrates the related modules. For reporting issues and requesting features, OpenCV has an issue tracking system ¹. To build Pairika, we reviewed 247 “Closed” issues that were labeled as “Bug” or “Feature”. Among the reviewed issues, we could extract 40 reproducible bugs that each yields at least one failing test. The extracted bugs are *isolated*, *reproducible*, and are related to the *source code*.

Pairika is publicly available at: <https://github.com/tum-i22/Pairika>

Table 3.5.: OpenCV Modules

Module	LoC	# of Files	# of Tests	# of Faults
Core functionality (core)	196550	345	10528	15
Machine learning (ml)	19398	34	39	5
Calibration and 3D reconstruct (calib3D)	44647	84	87	2
2D features (features 2D)	60557	101	119	1
Deep neural network (dnn)	147671	162	122	13
Video analysis (video)	13895	48	68	1
Computational photography (photo)	10880	47	166	3

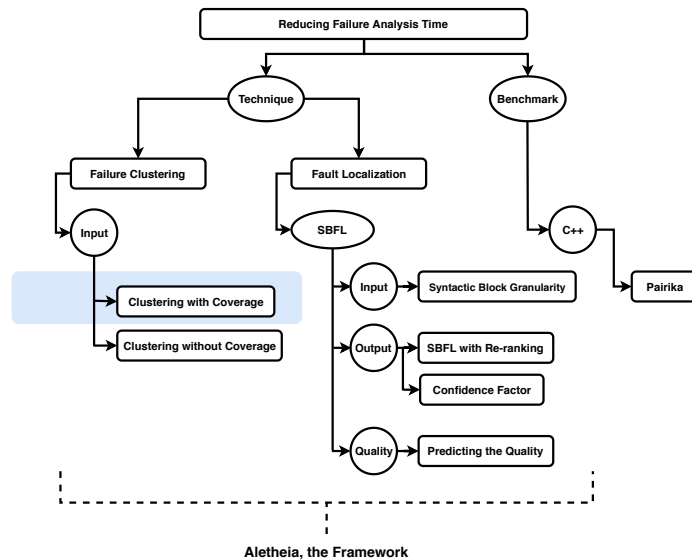
¹<https://github.com/opencv/opencv/issues>

Part II.

Failure Clustering

4. Failure Clustering with Coverage

This chapter presents a clustering approach to group failures based on their root-causes. In addition, it presents a methodology to apply failure clustering in a real-world context. Parts of this chapter have previously appeared in a publication [58], co-authored by the author of this thesis.



In this chapter, we explain our first clustering approach which uses coverage data. Thus, for evaluation, we used LCM SiL data set where test execution time is high and the source code is accessible. We devise a methodology for adapting Jones' debugging in parallel technique [77] as a clustering technique (see Chapter 2) to a real context. We augment this approach by a method for selecting representative tests. To analyze failures, rather than investigating all failing tests one by one, developers inspect only these representatives. Our evaluation results show that utilizing our clustering tool, developers can reduce failure analysis time by more than 80%.

4.1. Introduction

In this chapter, we tackle a problem which was identified while conducting industry-oriented research with LCM for testing software releases of car ECUs. The **problem** is the following. There are many TCs at the SiL level of testing. These tests are frequently executed as regressions happen. In case of failures, there are usually only a few underlying faults that cause a large number of failures. Considering the significantly high execution time of the tests, it is not always possible to find the first fault, resolve it and re-run all the tests. One current process in practice is investigating all failing tests to find the faults, removing the faults and re-running the tests. This approach makes debugging a very time-consuming process. How can we reduce failure analysis time?

One **solution** is to cluster failing TCs with respect to the fault that caused them, select one representative for each cluster, investigate only the representatives, find the underlying faults and resolve them, then re-run the tests.

Thus, in this chapter, we:

- propose a methodology for adapting the idea of debugging in parallel [77] to a real context, including an approach to choosing adequate parameter values and a tailored approach for measuring the quality of clustering. Our approach is methodologically different from [77] in the way that we propose an approach for segregating between failing tests rather than locating faults.
- propose a method for selecting representative tests to start the debugging process. Analyzing only the representative tests removes the need for analyzing all the failing tests one by one.
- investigate the effectiveness of our proposed approach in LCM SiL data set with ca. 850 KLOC. We also show how available software-related information at the SiL level can be used to analyze failures at the HiL level. Finally, we suggest new metrics for the effectiveness of TC clustering.

The above-mentioned contributions are the main differences between our work and [77].

4.2. Methodology

In the following, we describe our methodology. We first explain the first clustering technique introduced by Jones et al. [77] that we use as part of our approach, while completing it with our two additions that are selecting representative tests and utilizing SiL information for analyzing HiL failures. Then, we explain our strategy for gathering different methods and metrics and identifying the best ones.

4.2.1. Approach

First, we run a test suite and extract an execution profile for each TC. Second, utilizing agglomerative hierarchical clustering [154], we build a tree of failing tests based on the similarity of execution traces. In order to cut this tree into clusters we need to know the best number of clusters. In the third step, we hence utilize fault localization techniques to decide on the best number of clusters. Then, we cut the tree into the found number of clusters. Finally, in the fourth step, we calculate the centers of the clusters and choose the failures which are closest to the centers as representative tests. The tool we implemented is meant to be used as an assisting tool for developers. Therefore, they receive the list of representatives to investigate them. Steps one, two and three are taken from [77]. However, we have improved these steps to make them applicable in real contexts. In the following, we describe the steps and our improvements in detail.

Step 1: Running Tests and Profiling Executions

The first step is to run the tests and profile executions. To profile TC executions, we instrument the code using an open source tool for measuring code coverage for C++ and C.¹ Executing a program using this tool results in a report about which lines of code have been executed. The main advantage of this tool in our case is its compatibility with the Visual C++ compiler used in Microsoft Visual Studio, which, in the context of our case study, is the development tool used by our industry partner.

We developed a tool called Aletheia (see Chapter 10). The *Data Generation* functionality of Aletheia consists of three stages. The first stage is to prepare test execution according to the startup parameters and the system under test to be analyzed. The next stage involves the execution of the tests while recording the coverage information and the test results using the coverage tool. The third stage is optional and depends on the level of abstraction at which the analysis should happen. Since the tool we use provides statement-level coverage only, for function-level coverage analysis the third step is needed to partially parse the source files to aggregate the statement-level information to function-level coverage. The coverage information is the execution profile and is fed to the next step as input. In our case study, since controlling the hardware is in the form of functions to read/write signals, it is important to use function-level profiles.

Step 2: Generating Failure Tree

As described in Chapter 2, we utilize hierarchical clustering to generate a dendrogram of failing tests. We use execution profiles generated in the previous step, as our feature sets for clustering.

¹OpenCppCoverage, available at <https://opencppcoverage.codeplex.com/>, licensed under the GNU General Public License version 3 (GPLv3).

Step 3: Cutting the Failure Tree by Fault Localization

We use Spectrum-Based Fault Localization (SBFL) (explained in Chapter 2) to find the best number of clusters k , or the cutting point, of the dendrogram. Considering Figure 2.1, the question is which of the dashed horizontal lines 1 to 4 is the best cutting point. Liu and Han [109] as well as Jones et al. [77] suggest that if the failures in two clusters identify the same entities as faulty entities, they most likely failed due to the same reason and should be merged into one cluster.

This process can also be considered in a top-down manner. This technique computes the *fault localization rank* for the children of a parent to decide whether the parent is a better cluster or it should be divided into its two children. The result of fault localization result is a ranked list of entities from the most to the least suspicious. To check similarity between ranked lists, we used the Jaccard [133] set similarity as suggested by [77], defined on two sets A and B as follows.

$$\text{Similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (4.1)$$

If the similarity of the fault localization rank of two children is smaller than a predefined threshold, they are (likely) pointing to different faults. They are dissimilar and should not be merged. Thus, the parent cluster is not a good stopping point and should be divided into its children. Otherwise, the parent is a better cluster and this is the stopping point for clustering. According to Figure 2.1, the first step is to decide whether dashed line 1 is a better cutting point or dashed line 2. To answer this question, the fault localization rank at cluster c2 is compared to fault localization rank at cluster c3. If the similarity between these two sets is larger than the predefined threshold, they are similar and the parent c1 is a better clustering than dividing it into two clusters c2 and c3. As the result, line 1 is the cutting point. If line 1 is not the cutting point, the process continues to the point that no more division is needed. We somewhat arbitrarily selected 0.85 as the similarity threshold in our evaluation tests, and because of the good results, did not see a need to question this choice.

Step 4: Selecting Representative TCs

We have already grouped TCs based on their hypothesized root causes by generating the failure tree and cutting it into clusters. Now, we need to suggest a representative for each cluster. Developers investigate only the representatives to find all the faults. Since it is likely that clustering is imperfect, the selection of representatives for a cluster has a great importance: We are aware that clusters are unlikely to be 100 percent pure [38].

We require our solution to make the representatives reliable. To avoid selecting an outlier (a failure which does not belong to the fault class that has the majority in the cluster) as a representative in clustering, we hence calculate the center of the cluster and find the k -nearest neighbors (KNN) [25] to the center. These KNNs are selected as representatives of

the respective cluster. KNN search finds the nearest neighbors in a set of data for each data point. Based on discussion with test engineers, we considered $k=1$ in our experiment.

Step 5: Utilizing SiL Execution Profiles for HiL Analysis

For analyzing failures at the HiL level, we need to perform an extra step. Developers typically generate SiL and HiL tests from the same source. With SiL simulation, the software component is tested independently from the target hardware.

However, in our context, the information used for SiL can be re-used for the subsequent HiL tests. At the HiL level, real hardware components are integrated in a simulated environment for testing. Thus, there is a common system model between SiL and HiL. The difference is the use of mocked values instead of real hardware on SiL testing. Based on this information, it seems intuitive that if we do not have access to the whole execution profile of the HiL tests (containing both software and hardware parts), we are able to analyze failures at this level utilizing execution profiles from the SiL executions. Note that in those cases where some tests pass on the SiL level but fail on the HiL level, failure analysis is more difficult and time consuming.

Consider a TC used both at the SiL and HiL levels. First, we generate its execution profile by instrumenting the code at the SiL level. Second, we run the test at the HiL-level and get the verdict (failing/passing) information. Then, we combine these two to generate the feature set for hierarchical clustering and hit spectrum for fault localization as described in step 1. The rest of the steps will be the same. Therefore, we are able to analyze SiL and HiL failures using essentially the same information.

Table 4.1.: List of Mutations Applied [68, 129, 4]

Mutations
Negate condition in "for" or "while" statement
Replace integer constant by another integer
Delete a statement
Replace an operator by another operator
Assign null in assignment statements
Replace return expression with null

Table 4.2.: Average Values of Cophenet Correlation Coefficient on Training Data

	Average	Centroid	Single	Weighted	Complete
Euclidean	0.94	0.93	0.93	0.94	0.88
CityBlock	0.89	0.89	0.88	0.89	0.80
Minkowski	0.94	0.93	0.93	0.93	0.94
Cosine	0.90	0.89	0.87	0.89	0.87
Correlation	0.90	0.89	0.89	0.89	0.87
Jaccard	0.90	0.90	0.89	0.90	0.88

4.2.2. Parameter Setting

We mentioned in Chapter 2 that there is a plethora of similarity metrics and methods available in hierarchical clustering and SBFL. We need to select metrics with the highest performance in our context. We set up a training phase that helps us select the right metrics and parameters for this reason. We had access to 25 software components for different ECUs of a car. These software components have several sub-projects. We somewhat arbitrarily selected 100 sub-projects, 4 sub-projects from each software component, and injected faults in their source codes. Table 4.1 shows the list of mutations applied. We generated 100 first-order mutated versions and used these 100 faulty versions for our training data set. Although the external validity of using mutants in assessing the testing techniques has raised concerns, Andrews et al. [5] as well as Namin and Kakarla [129] suggest that under specific circumstances, replacing real faults with mutants has no significant impact on results. However, any analysis or generalization should be justified according to the influential factors including mutation operators, test suite size and programming languages. We applied the mutation operators previously used in the literature for fault localization [68, 129, 4]. To mitigate external threats, we then evaluated our approach using real faults.

Clustering Metric and Method Selection

We utilized the Cophenetic correlation [158] to compare the performance of different combinations of the selected methods and metrics explained in Chapter 2. Cophenetic correlation is a measure of “how faithfully a hierarchical tree preserves the pairwise distances between the original unmodeled data points or objects” [158].

Table 4.2 shows the average Cophenet value for clustering on 100 programs. In this experiment we clustered all the TCs (no matter if their verdict is passing or failing) based on their execution profiles’ similarity. The goal was to find the best metric and method for our context. Therefore, it was sufficient to measure how precisely the clustering tree represents the distance between TCs at this point. By looking at any clustering tree, it is possible to measure the distance between data objects. In a good clustering tree, these distances should be similar or close to the original distances between unclustered data objects. The results show that there is not a huge difference between different combinations of methods and metrics. Thus, we chose to utilize Average and Euclidean distance that are showing slightly better results for our context. Higher coefficients mean more accurate hierarchical trees.

Fault Localization Metric Selection

There are many different approaches for fault localization in the literature. We focused on surveys that compare and evaluate different metrics for SBFL. Lucia et al. [113] did a comprehensive study of association measures for SBFL. Wong et al. [185] introduced a new metric named DStar (D^*) and investigated the effectiveness of it in comparison with 31 other similarity metrics for locating bugs. The results of different evaluations

Table 4.3.: List of Implemented Spectrum-Based Fault Localization Metrics [185, 113]

Metric	Metric	Metric
Braun-Banquest (M ₁)	Odd Ratio (M ₂₅)	Two Way Support (M ₄₉)
Dennis (M ₂)	Yule's Q (M ₂₆)	Two Way Support Variation(M ₅₀)
Mountford (M ₃)	Yule's Y (M ₂₇)	Loevinger (M ₅₁)
Fossum (M ₄)	Kappa (M ₂₈)	Sebag-Schoenauer (M ₅₂)
Pearson (M ₅)	J-Measure (M ₂₉)	Least Contradiction (M ₅₃)
Gower (M ₆)	Support (M ₃₀)	Odd Multiplier (M ₅₄)
Micheal (M ₇)	Confidence (M ₃₁)	Example and counter- example Rate(M ₅₅)
Pierce (M ₈)	Laplace (M ₃₂)	Zhang (M ₅₆)
Baroni-Urbani (M ₉)	Conviction (M ₃₃)	Sorensen-Dice (M ₅₇)
Tarwid (M ₁₀)	Interest (M ₃₄)	Anderberg (M ₅₈)
Ample (M ₁₁)	Platesky-Shapiro's (M ₃₅)	Gini Index (M ₅₉)
Phi (M ₁₂)	Certainty Factor (M ₃₆)	Rogers and Tanimoto (M ₆₀)
Arithmetic Mean (M ₁₃)	Added Value (M ₃₇)	Ochiai II (M ₆₁)
Cohen (M ₁₄)	Collective Strength (M ₃₈)	Rogot2 (M ₆₂)
Fleiss (M ₁₅)	Klosgen (M ₃₉)	Hamann (M ₆₃)
zoltar (M ₁₆)	Information Gain (M ₄₀)	Sokal (M ₆₄)
Harmonic Mean (M ₁₇)	Coverage (M ₄₁)	Rogot1 (M ₆₅)
Simple-Matching (M ₁₈)	Accuracy (M ₄₂)	Kulczynski (M ₆₆)
Hamming (M ₁₉)	Leverage (M ₄₃)	Goodman (M ₆₇)
Scott (M ₂₀)	Relative Risk (M ₄₄)	DSatr (M ₆₈)
Dice (M ₂₁)	Interestingness Weighting Dependency(M ₄₅)	DStar2 (M ₆₉)
Jaccard (M ₂₂)	Goodman and Kruskal(M ₄₆)	DStar3 (M ₇₀)
Tarantula (M ₂₃)	Normalized Mutual Information(M ₄₇)	DStar4 (M ₇₁)
Ochiai (M ₂₄)	One way support (M ₄₈)	DStar5 (M ₇₂)

show that there is not a single metric that has the best performance for every program. It depends on the context, programming language [185] and also the fault density [38]. For our experiment, we picked 72 metrics from these papers to find the most effective one for our context. Table 4.3 shows the list of implemented metrics.

To evaluate the effectiveness of different metrics, we measure the average percentage of code needed to be investigated to locate faults. This evaluation approach has been used in many works such as [185] and [113]. As the result of fault localization, developers receive a ranked list of entities based on their suspiciousness score. Thus, first, they check the entity with rank 1. If that is a false positive, they continue with rank 2, 3 etc. to find the faulty entity. Therefore, the percentage of the code examined to locate the fault is one indicator for the effectiveness of the metric.

To reach our goal in this step, we used 72 metrics to locate faults in our 100 faulty programs. We measured the code examined for the best and worst cases and then calculated the median for each metric. Since we had 100 programs, to make comparison of results more straightforward, we calculated the average for each metric over all programs. In Figure 4.1, the x-axis shows the metrics (using their numbers presented in Table 4.3) and the y-axis shows the average percentage of the code examined.

Our results indicate huge differences between the effectiveness of different metrics. As the graph illustrates, the minimum value belongs to metrics number 71 and 72 which are DStar4 and DStar5. These metrics have the same base formula (see Chapter 2). The only difference is that in DStar5, the numerator is raised to the power of 5 rather than 4. DStar4 has fewer calculations. Therefore, we chose DStar4 as the fault localization metric.

Note that using the D^4 metric, a developer would be required to check about 40% of the code in average to locate the fault. This number is not a convincing result. Thus this technique cannot be used as an effective *fault localization technique* in our case. However, it is worthwhile to notice that we are using this technique as a *similarity measure rather than a debugging technique*. Thus, we will not get a penalty if the similarity metric that we use is unable to pinpoint the exact fault location in the first rank in some cases. Nevertheless, it is important to use a good similarity metric for this reason since a bad metric may show similar rank for a great percentage of entities of the code and this makes distinguishing between different failing executions difficult.

4.3. Experiment

We can now refine the research questions and describe evaluation results and threats to validity.

4.3.1. Research Questions

We aim to reduce failure analysis time. To this end, we cluster failures. We need to answer the following questions to evaluate how successful our solution is in reducing analysis

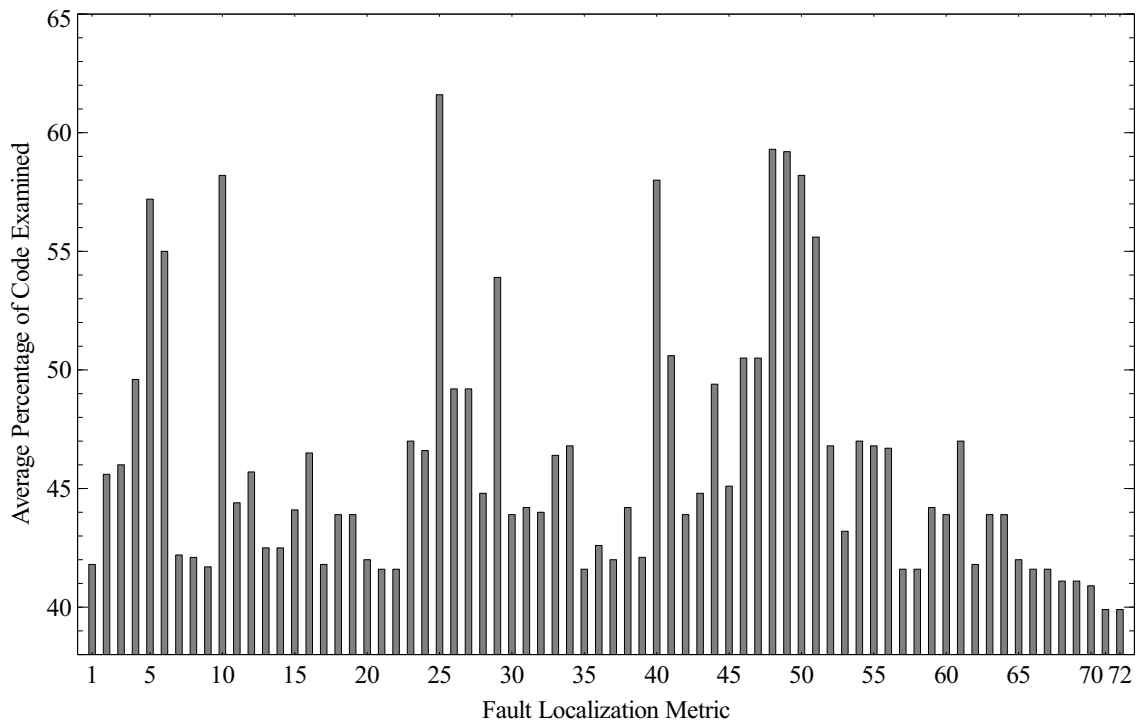


Figure 4.1.: Evaluation Results of Implemented Fault Localization Metrics

time:

RQ1: Can our clustering tool effectively serve as an assisting tool for developers to reduce the analysis time?

RQ2: How much reduction in time is achievable using clustering?

RQ3: Which metrics and clustering methods are most effective in our context?

Technically speaking, these questions boil down to determining how “well” our clustering schema works. We start by defining what “well” means.

4.3.2. Results

To evaluate the effectiveness of our approach in reducing failure analysis time, we conducted a study on LCM SiL builds introduced in Chapter 3. The results of the clustering evaluation are shown in Table VII. We are able to achieve high scores of purity and accuracy. In an ideal solution the number of clusters equals the number of underlying faults. The low scores of recall and thus F-measure and high scores of entropy show that the number of clusters are usually larger than the number of faults. Nevertheless, the numbers in the two last columns show that even though we do not always find the ideal number of clusters, we are able to achieve a huge reduction in analysis time anyway.

F-measure and entropy are measured and included for comparison purposes with existing work. However, we believe these numbers need to be interpreted with care. For example, 0.26 F-measure in SWC1-OldVersion2 does not sound convincing but it corresponds to a reduction of 95.41% (in the number of test cases, which we tacitly assume to be proportional to the time needed to debug). Also, in the same case, we were able to achieve 96.61% of the maximum possible reduction. This means that having to investigate 11 representatives instead of the ideal 3, we have lost only 3.39% (100-96.61) more reduction in time. The reason is the large number of TCs (1103) in this case. We would also conjecture that from a practical perspective, ARed may be easier to interpret than F-measure and Entropy.

When testing SWC1 and SWC2, some tests failed at the HiL but passed at the SiL level. This is not uncommon in practice. We generated the execution profiles based on SiL execution and attached the HiL verdicts to them. Then, we added these failures to the rest of the failures at the SiL level and clustered all the failures. 100% resp. 88% purity and 87% resp. 100% as achieved reductions in analysis time for these two cases show that the idea of matching SiL executions with respective HiL verdicts is successful and promising.

Based on the above results, we answer the research questions as follows:

RQ1: Considering purity and ARed scores, we believe that our clustering tool can serve as an effective assisting tool for developers. Utilizing this tool, they do not need to investigate all the failing tests one by one. Instead, they check the representative tests to find the underlying faults. In all the evaluation tests, the representative selection was successful in the sense that selected representatives were always failing due to the fault that had the majority in the cluster.

We believe that the observed high scores of purity and the proper representative selection method make our tool a reliable assistant.

Table 4.5 shows the number of failures per fault in each test suite. These numbers show that although in some cases we had highly imbalanced data, our clustering approach yielded high accuracy and was able to distinguish between all the distinct faults.

RQ2: Compared to the current process of debugging, utilizing our clustering tool can help in saving more than 80% of the failure analyzing time (where, once again, we equate the number of test cases with the time needed to analyze them).

RQ3: In our context, we found that the combination of “ D^4 as fault localization metric” and “Euclidean distance as distance metric and Average as clustering method” yielded the best performance for the chosen granularity of “functions” as entities for execution profiling.

To answer this question, we set up the training phase, which we argue can be reproduced for a different class of systems on the grounds of existing software components. One important point in our case is that we cannot use any clustering method that needs a predefined number of clusters or an upper threshold for the number of clusters. The number of clusters means the number of underlying faults which is unknown a-priori. Depending on the software component, development stage, level of testing, granularity of the execution profile, we may get different results.

4.3.3. Threats to Validity

Our results show that we could achieve a great reduction in failure analysis time by clustering the failing tests. Nevertheless, we do not claim that our experiment setup and results will be valid for all other contexts. We offered a methodology for adapting this idea to different industrial contexts.

Test suite quality has a great impact on fault localization [11] and therefore clustering. If all the test cases in a test suite cover the same parts (or most parts) of the code, it will be a difficult task to distinguish different failing (in the sense of different underlying causes) executions. There are results that show the threats to validity of spectrum-based fault localization [167] and its shortcomings [119]. However, we are not employing this technique for debugging and localizing faults, but rather as a similarity measure to better understand which failures happened due to the same reasons.

Table 4.5.: Number of Failures per Fault for Each Test Suite

	Number of failures per fault(F)
SWC1-OV1	F1:21 F2:3
SWC1-OV2	F1:26 F2:202 F3:12
SWC1*	F1:14 F2:3 F3:1 F4:3 F5:1 F6:8 F7:1 F8:1
SWC2*	F1:4 F2:14 F3:6 F4:1
SWC3	F1:7
SWC4	F1:1 F2:17 F3:14 F4:1 F:6
SWC5	F1:2 F2:27 F3:1
SWC6	F1:16 F2:1 F3:1 F4:1
SWC7	F1:66
SWC8	F1:2 F2:5 F3:1 F4:1
SWC9	F1:8

Another important factor is the granularity of profiling elements. We chose function-based coverage information both as object features in hierarchical clustering and for generating hit spectrum in fault localization based on the nature of our context. Using more fine-grained or coarse-grained entities may impact the performance.

Our first assumption was that each failure has only one reason behind it. Even though this assumption might not always hold true, we did not encounter any other case in our experiment. This may be because of the quality of the test suites as well. We are aware that there may be some cases where two or more faults cause the same failure.

The similarity threshold used in comparing fault localization ranks may also affect the results. Selecting a smaller number may result in a fewer number of clusters and selecting a larger number may result in a greater number of clusters. Therefore, this value can also impact the trade-off between purity and the number of clusters. We arbitrarily chose 0.85. The results show that it is a suitable threshold.

Focusing on one specific domain and one specific kind of ECUs is a potential threat to our external validity. It is unlikely that a general solution can be found for grouping failures

Table 4.4.: Evaluation Results - Performance = 0.4 * FoundCauses + 0.2 * Purity + 0.4 * ARed

Build	#Failures	#Faults	#Clusters	Precision	Recall	F-measure	Entropy	FoundCauses	Purity	ARed	Performance
1	24	2	8	1	0.34	0.51	1.36	1	1	0.72	0.888
2	240	3	11	0.96	0.15	0.26	3.12	1	0.98	0.96	0.98
3*	32	8	11	1	0.74	0.85	1.32	1	1	0.87	0.94
4*	25	4	4	0.76	0.84	0.8	0.93	1	0.88	1	0.976
5	7	1	1	1	1	1	0	1	1	1	1
6	39	5	6	0.43	0.58	0.48	0.98	1	0.77	0.94	0.93
7	30	3	3	0.92	0.99	0.96	0.69	1	0.97	1	0.994
8	19	4	4	1	0.8	0.89	0.69	1	1	1	1
9	66	1	2	1	0.96	0.97	0.042	1	1	0.98	0.992
10	9	4	6	1	0.42	0.59	1.07	1	1	0.60	0.84
11	8	1	1	1	1	1	0	1	1	1	1

in different contexts. We performed the training phase on real industrial data. We used all available software components of car ECUs. Nevertheless, we cannot claim the external validity of our results. Applying the same technique, parameters, and values in different contexts might produce different results.

As mentioned earlier, fault injection gives rise to threats to external validity. Recent works [68, 83] suggest that real faults are replaceable with mutations if mutations are used with caution and if they are representative of real faults. In our work, we used some of the mutation operators previously applied in the literature to learn the best metric and distance measure. For the evaluation, we used real world systems and real-world faults.

4.4. Conclusion

One challenge in testing automotive CPSs is reducing failure analysis time. We built on the idea to cluster failing tests to reduce the effort needed by developers and investigated its effectiveness in LCM SiL data set with ca. 850 KLOC. Results show that we can group failures based on their underlying faults with very high purity. The clustering tool can effectively reduce the effort needed by developers. Utilizing this tool, developers save more than 80% in failure analysis time, at least in our study.

We believe that our results add to the growing body of evidence about the potential usefulness of fault localization techniques if multiple faults are present. Complementing earlier work, our study indicates that these techniques may be more useful for organizing the debugging process than for actually locating faults. To the best of our knowledge, we are the first to provide this evidence with a large industrial case study for embedded automotive systems.

Our methodology is meant to be instantiated to a specific domain or class of applications. We have shown how to learn the relevant parameters (e.g., clustering method, distance measure) as a significant part of the initialization process. While we cannot report on a second study in a different domain here and clearly see the threats to external validity, we ourselves are sufficiently confident to repeat our own study in a different context.

5. Failure Clustering without Coverage

This chapter presents a failure clustering approach that does not need coverage data. Parts of this chapter have previously appeared in a publication [57], co-authored by the author of this thesis.



We saw in the previous chapter that to effectively reduce the analysis effort, the clustering tool selects a representative test for each cluster. Instead of analyzing all failing tests, developers only inspect the representative tests to find the underlying faults. In this chapter, we propose a clustering technique to group failing tests based on non-coverage data, retrieved from three different sources. We evaluated the effectiveness and efficiency of our solution using LCM HiL data set where source code is not accessible. The results show that utilizing our clustering tool, developers can reduce the analysis time more than 60% and find more than 80% of the faults only by inspecting the representative tests.

5.1. Introduction

According to Chapter 4, one remedy to the challenge of excessive analysis effort is using a pre-analysis method that clusters failures with regards to their underlying root causes. The

results of the test execution can be passed on to the clustering tool, which generates and returns a set of clusters and one representative for each cluster. Each cluster would contain only one root cause in an ideal clustering, and all the root causes would be covered by the chosen representative TCs. The reduction of the analysis effort can then be achieved by analyzing only the representative tests rather than all the failing tests.

However, in Chapter 4, we focused on clustering failing tests at the SiL level, where software components are tested within a simulated environment without any actual hardware. We used the coverage profile of tests as the input for their clustering tool. It is not always possible to use this kind of data in practice due to three reasons. First, the source code is not always accessible (e.g., in the case of HiL tests). Second, in this approach, collecting coverage information when running passing tests is also needed. This requirement is unnecessary and imposes extra work on the system. Third, instrumenting very large projects when running integration tests can be very expensive and time-consuming.

The other similar existing approaches in the literature (see Chapter 9) are either coverage based or use context-specific data. To bridge this gap, we propose a clustering tool which groups failing tests based on different *code independent* data. This tool can be used in different levels of testing, e.g., HiL, SiL, etc.

Thus, in this chapter, we solve a **problem** similar to the problem explained in Chapter 4. As another **solution**, we propose to cluster failing TCs with respect to their root causes but without coverage data. We select one representative for each cluster. Investigating only the representatives, lead us to find all the root causes without having to inspect all the failing tests.

Thus, in this paper, we:

- Propose a new set of non-code-based data to cluster failing tests. These new data make the clustering tool applicable in different levels of testing and different contexts.
- A clustering tool that can also be used a priori for test selection and prioritization. Since for each test run, due to time and resource constraints, only a subset of tests can be executed, a clustering tool can help in grouping tests based on their similarities and introducing the cluster representatives for the next test run. This can raise the diversity in each test run.
- Propose a methodology for adapting the clustering idea to a real context. We investigate the effectiveness of the adapted approach in LCM HiL data set with ca. 1.3 million LoC and 13000 tests.

5.2. Approach

In the following, we describe our methodology. We use the clustering approach explained in Chapter 4 as the base of our work. Then, we suggest the data sources that make this clustering approach applicable in different stages of testing and other purposes such as test prioritization.

First, we collect the data from different non-code-based sources, e.g., Jira tickets to make a feature vector for each test case. We binarize all of them to prepare them for hierarchical clustering. Second, utilizing agglomerative clustering, we build a tree of failing tests based on the similarity of their feature vectors. Third, using a regression model on the number of failing tests in previous test runs, we predict the number of clusters. Finally, in the fourth step, we calculate the centers of the clusters and choose the failing tests which are closest to the centers as the representative tests. The developers receive the list of representatives to investigate them. If the suggested number of clusters appears to the developers to be inaccurate, they can immediately adjust the number of clusters on the user interface and get new representatives. Steps two and four are taken from Chapter 4. In the following, we describe all the steps in detail.

5.2.1. Generating the Input Data

Two main input data sources are: the database of test results that includes several thousand test results from the previous test runs, and the repository of the TCF test cases, providing the source files for the tests. We could extract five sets of features (variables in a data set) using these two data sources. Since the primary objective of this paper is to cluster failing tests, these data are extracted only for failing tests. In case of test selection or prioritization, they can be extracted for all tests. Typically, multiple projects (e.g., weekly, daily, nightly) are used to test a single ECU. Each test run is called a *build*. All the feature values are extracted individually for each build. We explain each set of features in the following.

General Features. We extracted the following features from the database [102]:

- **T.Id:** specifies the test identifier.
- **Agent:** specifies the name of hardware which is executing the tests.
- **Component:** specifies the component of the BDC the test belongs to.
- **Domain:** specifies to which domain of the BDC the test belongs to.
- **File:** specifies from which underlying TCF file the test was generated. A single TCF source file may be used to generate multiple test cases.

Table 5.1 shows an example of data values for these features. As all these features are of categorical nature and do not follow an ordinal scale, we transformed them into binary data.

Failed/Passed History. To exploit the knowledge of historical test executions, we generated Failed/Passed history matrices. The general idea of this feature set is to express the similarity between two test cases based on the times they shared the same execution result. Naturally, a test result can either have the status *Fail* (F) or *Pass* (P). However, in practice due to the resource constraints, it is not guaranteed that every test case is executed in every build. Thus, we added another value as *Not executed* (N) [102].

Table 5.1.: General Features [102]

Test	Agent	Domain	Component	File
TC 1	Agent-1	Comfort	SWC1	Audio.tcf
TC 2	Agent-2	Body	SWC2	FesModes.tcf
TC 3	Agent-1	Body	SWC2	FesModes.tcf

Table 5.2.: Fail/Pass History [102]

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	P	P	F	P	F
TC 2	P	N	N	F	N
TC 3	P	N	F	P	P

Table 5.2 demonstrates an example. Again, we need to transform it to a binary table; thus we generate two tables, one to represent the similarity based on co-occurrences of Failed statuses and one to represent the similarity based on co-occurrences of Passed statuses. As an example, Table 5.2 has been transformed to the Tables 5.3 and 5.4 [102].

Table 5.3.: Binary Failed History

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	0	0	1	0	1
TC 2	0	0	0	1	0
TC 3	0	0	1	0	0

The Failed/Passed distance d_{fp} between two tests x and y can therefore be defined as [102]

$$d_{fp}(x, y) = 0.5 * d_f(x, y) + 0.5 * d_p(x, y) , \quad (5.1)$$

where $d_f(x, y)$ denotes the distance between two tests based on failing history and $d_p(x, y)$ denotes the distance between two tests based on passing history using any specified distance metric (see Section 2.1.1).

Broken/Repaired History. Similar to the Failed/Passed History, the Broken/Repaired History aims at exploiting historical execution knowledge, with a small difference. In the Broken/Repaired History the focus lies on the transition of test statuses. A transition from status *Failed* to status *Passed* is defined as a *Repaired* event (R), whereas a transition from status *Passed* to status *Failed* is defined as a *Broken* event (B) [102]. Following this definition, an arbitrary Failed/Passed History can be transformed into a Broken/Repaired History. If no transition takes place, e.g., if a test case failed in two successive builds, a *No Event* (N) label is used to fill the gap. Table 5.5 shows an example.

Following the approach described for the Failed/Passed History, again two binary tables should be generated, one to represent the similarity based on co-occurrences of Broken

Table 5.4.: Binary Passed History

Test	Build.1	Build.2	Build.3	Build.4	Build.5
TC 1	1	1	0	1	0
TC 2	1	0	0	0	0
TC 3	1	0	0	1	1

Table 5.5.: Broken/Repaired History [102]

Test	Build.1	Build.2	Build.3	Build.4	Build.5
TC 1	N	N	B	R	B
TC 2	N	N	N	N	N
TC 3	N	N	B	R	N

events and one to represent the similarity based on co-occurrences of Repaired events. The computation of distances for two failing tests is similar to the Failed/Passed History calculations.

Jira History. We used the Jira tickets to extract the next feature set which is based on the faults assigned to the previously analyzed failed tests. The idea is that tests which frequently shared the same cause in the past are also likely to fail due to the same cause in the future [102]. Table 5.6 shows an example. Each “cause” is a Jira ticket ID that has been assigned to the failing test. One ticket may be assigned to several failing tests if the manual analysis shows that these tests are failing because of the same reason. Similar to the previous feature sets, this table should change to a binary form as shown in Table 5.7.

Table 5.6.: Jira History [102]

Test	Project	Build	Cause
TC 1	project 1	build 1	cause x
TC 2	project 1	build 1	cause x
TC 3	project 1	build 1	cause y
TC 1	project 1	build 2	cause z

TCF Test Case Similarity. TCF files used to generate TCs as described in Chapter 3 are maintained in SVN repositories. These repositories are referenced to define the source files needed to generate the desired test series. Our hypothesis is that the likelihood that two tests failed due to the same cause increases with the similarity of their underlying TCF source files [102]. To facilitate the calculation of similarity between two TCF files, we translate TCF files into a standard machine-readable format (JSON). Figure 5.1 shows an example JSON output for the `Audio.Example.tcf` file, introduced in Chapter 3, Figure 3.3.

Using this JSON representation, the similarity between source files of any two test cases can be calculated using a combination of the following features [102]:

Table 5.7.: Binary Jira History

Test	Build1CauseX	Build1CauseY	Build2CauseZ
TC 1	1	0	1
TC 2	1	0	0
TC 3	0	1	0

```

{
  "tcName": "Audio_Example",
  "codingSteps": [
    {"codingStepName": "BE_AUDIO_VERB",
     "codignStepValue": "01"},
    {"codingStepName": "BE_AUDIO_LIN_VAR",
     "codignStepValue": "01"}
  ],
  "executionSteps": [
    {"executionStepActor": "PWF",
     "executionStepType": "SET",
     "executionStepBusType": "LIN",
     "executionStepBus": "KLIN22",
     "executionStepBusMessage": "0xA",
     "executionStepSignal": "ST_CON_VEH_CSG_LIN",
     "executionStepValue": 2},
    {"executionStepActor": "Audio_Volume",
     "executionStepType": "SET",
     "executionStepBusType": "LIN",
     "executionStepBus": "KLIN8",
     "executionStepBusMessage": "0x2A",
     "executionStepSignal": "ST_TURN_AUDCU_LIN",
     "executionStepValue": 15}
  ]
}

```

Figure 5.1.: JSON Representation of Audio_Example_1.tcf [102]

- **CODING:** Each Coding step in a test is mapped to a binary feature. For each test case, it is checked whether the test case contains the Coding step or not (true = 1, false = 0). Two Coding steps are regarded as equal if they share the same name and the same value.
- **ACTOR:** Each Actor in a test is mapped to a binary feature. For each test case it is checked whether the test case uses the given actor to perform an arbitrary action in any of its test steps or not (true = 1, false = 0).
- **ACTOR.TYPE:** Each Actor/Type combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to perform an action of the given Type (e.g., set or check) in any of the test's steps (true = 1, false = 0).

- **ACTOR_VALUE:** Each Actor/Value combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to set or check the given Value in any of the test's steps (true = 1, false = 0).
- **ACTOR_TYPE_VALUE:** Each Actor/Type/Value combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to perform an action of the given Type for the given Value in any of the test's steps (true = 1, false = 0).
- **BUS_TYPE:** Each Bus Type in a test is mapped to a binary feature. For each test case it is checked whether the test case performs an arbitrary action on a Bus of the given Type in any of its steps (true = 1, false = 0).
- **BUS:** Each Bus in a test is mapped to a binary feature. For each test case it is checked whether the test case performs an arbitrary action on the given Bus in any of its steps (true = 1, false = 0).
- **BUS_MESSAGE:** Each Bus Message in a test is mapped to a binary feature. For each test case it is checked whether the test case uses the given Bus Message to perform an arbitrary action in any of its steps (true = 1, false = 0).

As an example, Table 5.8 shows the binary values for two examples in Figures 5.1 (TC1) and 5.2 (TC2) considering only ACTOR and ACTOR_TYPE features.

```

{
  "tcName": "Audio_Other_Example",
  "codingSteps": [
    { "codingStepName": "BE_AUDIO_VERBAUT",
      "codingStepValue": "01" },
  ],
  "executionSteps": [
    { "executionStepActor": "Audio_Volume",
      "executionStepType": "SET",
      "executionStepBusType": "LIN",
      "executionStepBus": "KLIN8",
      "executionStepBusMessage": "0x2A",
      "executionStepSignal": "ST_TURN_AUDCU_LIN",
      "executionStepValue": 99 }
  ]
}

```

Figure 5.2.: JSON Representation of Audio_Example_2.tcf [102]

5.2.2. Clustering Failing Tests

As explained in Chapter 2, we use Hierarchical clustering since it enables users to retrieve an arbitrary number of clusters without the need to re-execute the clustering algorithm.

Table 5.8.: Binary TCF Similarity Features

Test	PWF	AudioVolume	PWFSet	AudioVolumeSet
TC 1	1	1	1	1
TC 2	0	1	0	1

This is especially useful in practice since in a real-world scenario, it will not limit the users to a single suggested number of clusters. Users will be able to explore multiple alternatives without the need to wait for the re-execution of the clustering tool.

Predicting the Number of Clusters

The number of predicted clusters relates to the number of underlying faults, therefore an accurate prediction is important for developers. We considered two regression models, Linear regression [86] and Polynomial regression [86], to examine the relationship between the number of failing tests and the cutting distance on the Hierarchical tree.

In our experiment, we extracted the real number of faults in the previous analyzed builds from the database. Then, we calculated the cutting distances of the respective trees. Finally, we fitted the regression models to predict the cutting distance based on the number of failing tests.

5.2.3. Selecting the Representatives

As mentioned earlier, the main goal of this paper is to reduce the analysis effort by selecting some representatives for failing tests, so that only the representatives should be analyzed instead of all the failing tests. To this end, we selected the clusters' medoids as representatives. A medoid is the object which is closest to the geometric center of the cluster [168]. Since we want the developers to be able to change the number of the cluster in real-time, we precompute the representatives for all the clusters considering all possible cutting distances.

5.3. Experiment

To evaluate how successful our clustering without coverage approach is in achieving its primary objective, reducing failure analysis time, we need to answer the following **research questions**:

RQ1. How effective are non-code-based data in clustering failing tests? How much reduction in analysis time is achievable? How many of the underlying faults are detectable?

RQ2. How efficient is the clustering tool?

RQ3. Which set of input features are the most important and useful in clustering failing tests?

Table 5.9.: Performance Values Using Centroid Method [102]

Feature Set	Dist. Metric	<i>Performance_{Avg}</i>
General Features	Hamming	0.82
Failed/Passed History	Hamming	0.76
Broken/Repaired History	Hamming	0.75
Jira History	Hamming	0.81
TCF Similarity	Euclidean	0.83

RQ4. Which distance metrics and clustering methods are the most effective in the given context?

Since there are several parameters in our approach that we need to set before applying it, we set up a training phase that helps us select the right metrics and parameters for this context. Thus, first, in the following, we explain the available data, evaluation metrics, and the training phase. Then, we explain the evaluation results in the next section.

5.3.1. Parameter Setting

We developed our clustering tool as a standalone Python application using SciPy ¹, NumPy ², and Pandas ³. Fitting our clustering model involves setting the following parameters:

1. the optimal clustering method
2. the optimal distance metric for each of five groups of features
3. the optimal weights for each of five groups of features
4. the optimal way of predicting the number of clusters

to find the best parameter for each item in the training phase, we used the *Performance* metric.

Clustering Method and Distance Metric

The Performance values of different method and distance metric combinations do not show a significant difference. Comparing all the Performance values, we chose “Centroid” as the clustering Method, and “Hamming” as the distance metric for General, Failed/Passed, Broken/Repair, and Jira features sets, and “Euclidean” as the distance metric for TCF Similarity feature set. Table 5.9 shows the average Performance values on the training data set.

¹<https://www.scipy.org/>

²<http://www.numpy.org/>

³<https://pandas.pydata.org/>

Table 5.10.: Best Performing Weights for Input Feature Sets [102]

Feature Set	Symbol	Value
General Features	$w_{general}$	0.29
Failed/Passed History	w_{fp}	0.01
Broken/Repaired History	w_{br}	0.09
Jira History	w_{jira}	0.31
TCF Similarity	w_{tcf}	0.30

Input Feature Weights

As described in Section 5.2.1, we extracted five set of features as input for our clustering. These five sets, lead to the generation of five different data sets. Since it is not clear which sets of features are more useful in clustering, we generated 1000 random weight combinations. To measure the similarity between two tests, first, we measure the similarity based on all the five feature sets. Then, assigning the weights to each group, we sum up the similarity values to find the combined similarity value. Table 5.10 shows the best combination of the weights. The results show that General Features and the Jira History, and TCF Similarity sets are the most important features, while Broken/Repaired History and Failed/Passed History are not so useful.

Predicting the Number of Clusters

We used Linear and Polynomial regressions to investigate the correlation between the number of failing tests and the cutting distance in the Hierarchical tree of failing tests (see Chapter 2).

The results show that applying Polynomial regression leads to a better performance than Linear regression, $P_{polynomial}=0.80$ and $P_{linear}=0.76$. Figure 5.3 depicts the fitted Polynomial regression model, where x is the number of failing tests, and the $f(x)$ shows the predicted distance to cut the tree.

5.4. Evaluation

We can now answer the research questions, describe evaluation results, and explain threats to validity.

5.4.1. Summary of Results

The results of the clustering evaluation are shown in Table 5.11. As the results show, we are able to achieve high scores of Purity, FoundCauses and also ARed. Since the number of predicted clusters is usually larger than the number of underlying faults, Recall, F-measure and Entropy metrics do not show high scores. However, we are able to achieve a huge

Table 5.11.: Evaluation Results - Performance = $0.4 * \text{FoundCauses} + 0.2 * \text{Purity} + 0.4 * \text{ARed}$ [102]

Build	# Failures	# Faults	# Clusters	Precision	Recall	F-measure	Entropy	FoundCauses	Purity	ARed	Performance
31	142	9	44	0.99	0.14	0.25	0.04	0.89	0.98	0.74	0.85
54	77	26	29	0.87	0.68	0.76	0.15	0.88	0.95	0.94	0.92
75	86	12	13	1.00	0.79	0.88	0.05	0.92	0.94	0.99	0.95
102	96	25	35	0.96	0.63	0.76	0.05	0.96	0.99	0.86	0.93
114	37	20	20	0.94	0.71	0.81	0.05	0.95	0.98	1.00	0.98
124	185	10	65	0.97	0.03	0.07	0.03	0.90	1.00	0.69	0.83
134	84	23	26	0.89	0.84	0.86	0.14	0.83	0.95	0.95	0.90
144	121	33	63	0.39	0.20	0.26	0.40	0.82	0.94	0.66	0.78
154	78	13	35	0.93	0.12	0.21	0.05	1.00	0.99	0.66	0.86

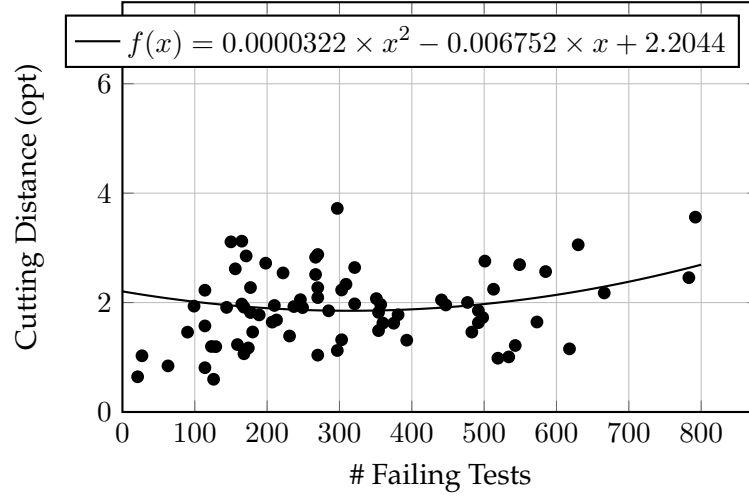


Figure 5.3.: Polynomial Regression Model for Cutting Distance [102]

Table 5.12.: Average Scores of Performance Metrics Using Different Weights [102]

Metric	Initial Setup	Alternative 1	Alternative 2
Purity	0.9674	0.9740	0.8989
ARed	0.8312	0.7990	0.9520
FoundCauses	0.9049	0.9151	0.6566
Performance	0.8879	0.8686	0.8338

reduction in analysis time anyway and find more than 80% of the causes by investigating only the representatives. For instance, in Build 154 with 78 failing tests, the ground truth says there are 13 underlying faults. We predicted 35 clusters that are 99% pure. These 22 extra clusters (35-13) lead to poor Recall (0.13) and consequently F-measure (0.22) scores. Nevertheless, with selecting one representative for each of these 35 clusters, and analyzing only them, we reduced the analysis time 66% $((1-(35/78))/(1-(13/78)))$ and found all the underlying faults (FoundCauses=1.0).

Based on the industry partner's needs and experts' advice, we assigned higher weights to the combination of FoundCauses and Purity (0.6) rather than the ARed (0.4). Due to this reason, the results tend to a larger number of clusters which are almost pure. Nevertheless, we have always achieved more than 60% reduction in the analysis time. Assigning higher weights to ARed in the training phase leads to different parameter setting and consequently different results. To compare the results considering different weights, we considered two alternatives: 1. $w_{fc} = 0.6$, $w_{ar} = 0.4$, and $w_p = 0.0$ and 2. $w_{fc} = 0.4$, $w_{ar} = 0.6$, and $w_p = 0.0$. Table 5.12 shows the average scores of performance metrics.

Based on the above results, we answer the research questions as follows:

RQ1. Considering the Performance metric which is a combination of FoundCauses, Purity, and ARed metrics, our idea in using non-code-based data for clustering failing test

is quite effective. In the evaluation, on average 90.4% of the faults are found when only analyzing the representative tests. Furthermore, an average Purity of 96.7% is achieved, which means that only 3.3% of the failing tests would be assigned to a wrong fault when assigning the root cause of the cluster's representative to all members of the cluster. The results show an average of 83.1% reduction in analysis time.

RQ2. Clustering based on non-code-based data is more efficient than clustering based on code-based execution profile (if the source code is available). In our approach, we need to compute the data for similarity comparisons one time a priori; then we can use them several times. Code-based techniques such as the one presented in Chapter 4 need to instrument the code which can be expensive in case of integration testing. Considering the without coverage approach, data collection and pre-processing took 5 hours while running the clustering itself took 3 minutes in the largest case. Collected data can be updated at longer intervals such as every two months.

RQ3. We set up a training phase to answer this question. We randomly generated 1000 weight coefficients for our five groups of feature sets. As shown in Table 5.10, General Features, Jira History, and TCF Similarity are the most important groups. There is not any significant difference between these groups.

RQ4. Like RQ3, we found the answer to this question in the training phase. Our experiment show that the impact of the clustering method and distance metric on the clustering performance is negligible. However, choosing a suitable strategy to determine the number of clusters is highly important.

Cross-validation. To test our model's ability to predict new data and to give an insight on how the model will generalize, in addition to the previous setting, we did 10-fold cross-validation. In our cross-validation setting, we used the training part to select the best weights for feature combination and to fit the Polynomial regression model. However, we preserved the distance metric and the clustering method unchanged since they do not have a significant impact on the results. The average scores are:

Foundcauses=0.8813, Purity=0.9700, and ARed=0.8813.

Considering $w_{fc} = 0.4$, $w_{ar} = 0.4$, and $w_p = 0.2$, these scores yield **Performance=0.8813**. For obvious reasons, we could not just report the cross-validation results. Since we could not report the average scores of metrics such as “# of Faults”, “# of Failures”, and “# of Clusters”, we selected a snap of data as the test data set to be able to show and discuss the results in Table 5.11.

5.4.2. Threats to Validity

We evaluated our approach in a large scale industrial case. For both training and evaluation, we used real-world systems, failures, and faults. Nevertheless, we do not claim that our experimental setup and results will be valid for all other contexts.

Proper data recording has a great impact on our clustering approach. We offered five feature types to measure the semantic similarity between test cases. However, if failure

analysis results, root causes information, patches and changes are not stored frequently in database, SVN, and Jira systems, the clustering results will not be accurate.

To find the best solution for our case, we set up a training phase. To set the parameters and assign the weights, we considered industry needs and experts' advice. Changing the parameters based on different needs could affect the final results. Nevertheless, we tried to propose a methodology for applying this idea in different environments.

5.5. Conclusion

Analyzing failing tests in the scope of automotive hardware-in-the-loop regression testing is a time-consuming task and requires a significant amount of manual work. To reduce the excessive analysis effort, we developed a clustering tool that groups failing tests with respect to their underlying faults and does not use coverage information. The analysis time is then reduced by proposing a single representative test for each cluster so that developers only have to analyze the representative tests instead of all failing tests. We suggested five different non-code-based data source to measure the similarity between failing tests. We evaluated the effectiveness and efficiency of our proposed idea in an automotive context with 86 regression test runs containing 8743 failing tests and 1531 faults.

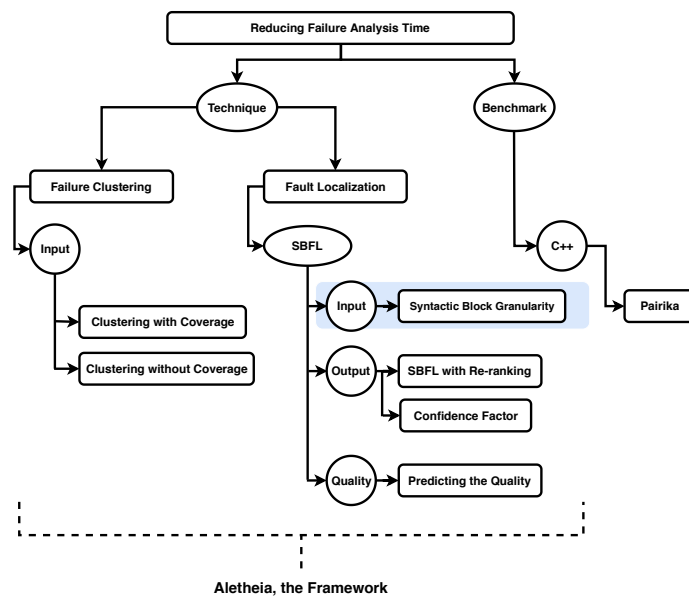
The results show that out of the five suggested data feature sets, Jira History, General Features, and TCF Similarity are the most useful features in this context. While clustering method and distance metric do not have a significant impact on the results, the technique used for predicting the number of clusters has high importance. We fitted a Polynomial regression model based on the number of failing tests to predict the number of clusters. With this approach, we are able to achieve more than 60% reduction in analysis time (an average of 83.1% in our experiments).

Part III.

Fault localization

6. Improving SBFL Through Using Syntactic Block Granularity

This chapter presents a new granularity level that helps in boosting spectrum-based fault localization effectiveness.



We saw in Chapter 2 that existing SBFL approaches are not applicable in practice yet. One problem is that these techniques rely on program spectra data that is either too fine- or coarse-grained [119]. In this chapter, we propose a new granularity of program spectra data, which takes into account the syntactic blocks present in the source code of the program. By evaluating our new syntactic block granularity on the bugs in Defects4J, we found that the syntactic block granularity exhibits best-case absolute ranking behavior similar to the method granularity (better than statement granularity), while having a wasted effort equivalent to, if not better than, the statement granularity (better than method granularity). Furthermore, it covers more types of faults than both existing granularities. Finally, when compared to the method granularity, it exhibits up to a 92.48% improvement when it comes to the locality of the program elements to the fault, a characteristic that provides the user with a better insight into the possible location of the faults.

6.1. Introduction

For SBFL techniques, the granularity of the program elements in the program spectra is important, not only to the effectiveness of the system, but also to the preferences of practitioners [94]. Kochhar et al. found that among surveyed practitioners, method, followed by statement and basic block were the most preferred granularities. But when it comes to the effectiveness of the system, the **problem** is that the method and statement granularities may be too coarse- or fine-grained, respectively, to properly locate the faulty program elements [119, 136]. Unfortunately, there is no golden rule to say which granularity is the best for all contexts. However, in an effort to combat these issues, as a **solution**, we propose a new syntactic block granularity which takes into account both coarse- and fine-grained program elements. A syntactic block can be defined as a block of statements that syntactically belong together to form a program element, such as a method declaration, or a while loop. By considering different types of syntactic components in the program's source code, we are able to capture a wide spread of faults, while providing a better insight into the possible location of the fault within the program element.

For the evaluation of our proposed syntactic block granularity, we used the Defects4J data set. Our results show that our proposed syntactic block granularity exhibits the advantage of method granularity over statement granularity, meaning a best-case absolute ranking behavior similar to the method granularity, while having the same advantage of statement granularity over method granularity, meaning a wasted effort equivalent to, if not better than, the statement granularity. Furthermore, it covers more types of faults than both existing granularities. Finally, when compared to the method granularity, it exhibits up to a 92.48% improvement when it comes to the locality of the most suspicious element e^* to the fault, a characteristic that provides the user with a better insight into the possible location of the fault within the given program element.

In this chapter:

- We propose a new granularity for program spectra called the syntactic block granularity which considers 18 different types of program elements.
- We evaluate our contribution using Defects4J. In doing so, we show that our syntactic block granularity covers more faults than both statement and method granularities, while having better wasted effort performance. Furthermore, it provides the user with a better insight into the possible location of the fault when searching through program elements.

6.2. Motivation

As mentioned previously, the two main program spectra granularities used by practitioners are statement and method [93]. Unfortunately, neither of these options are perfect, as they both have their limitations.

Due to its fine-grained nature, the statement granularity has a number of drawbacks. First, simple profile elements like statements cannot properly characterize and reveal nontrivial faults. Statements might be too simple to describe some complex faults, such as those that are induced by a particular sequence of statements [119]. In the Defects4J data set, the median size of a fault is four lines, with 244 bugs having faults spread across multiple locations in the program [164]. Furthermore, due to the nature of the statement granularity, it is incapable of locating bugs due to missing code, known as a fault of omission [182]. For example, of the 395 bugs in the Defects4J data set, only 228 can be localized by the statement granularity.

Unfortunately, it is unclear whether developers can actually determine the faulty nature of a statement by simply looking at it, without any additional information [182, 135]. As a possible solution to the drawbacks of the statement granularity, Masri suggests the usage of more-complex profiling types with higher granularity [119]. Previous empirical studies have shown that the effectiveness of SBFL techniques improves when the granularity of the program elements is increased [120]. For that reason, among others, many practitioners prefer to use the method granularity.

However, due to its coarse-grained nature, the method granularity has a handful of drawbacks when used for calculating SBFL scores. Sohn and Yoo suggest two drawbacks to the method granularity due to the nature of methods themselves [166]. First, methods on a single call chain can share the same spectrum values, resulting in tied SBFL scores. Second, if there are test cases that only execute non-faulty parts of a faulty method, they will decrease the overall suspiciousness score of the given method. Furthermore, when given a list of methods ranked by their suspiciousness, a programmer would still have to walk through all the statements in each method while looking for the bug, which can result in a lot of wasted effort, especially if the methods are large. Finally, the method granularity also lacks any sort of context and may not provide any further information to the developer. For instance, if there are failing test cases that focus on testing one specific method, such as a unit test, the developer will already know that the fault is contained within the failing method, so the method granularity results are of no additional help.

As both the statement and method granularities exhibit drawbacks, there is a clear need for a new granularity level that has a higher granularity than statements, without the added wasted effort and lack of context of methods. As a possible solution, we propose the usage of the syntactic block granularity. Based on different syntactic components found in the program's source code (see Table 6.4 for syntactic blocks in Java), it considers a wide range of program elements in an effort to provide more context to the developer with minimal added cost.

To illustrate the drawbacks of the statement and method granularities, as well as highlight the benefits of the syntactic block granularity, consider the sample program in Figure 6.1. The method *mid()* takes as input three integers and outputs the median value. The method contains a fault of omission, where the proper implementation should include the else-if block from lines 8-10. Tables 6.1, 6.2, and 6.3 contain the coverage information from a test suite containing six different test cases for each of the three different granularities. In each

```

1: public int mid(int x, int y, int z) {
2:     int mid = z;
3:     if (y < z) {
4:         if (x < y) {
5:             mid = y;
6:         }
7:         /* FAULT: missing the following
8:         else if (x < z) {
9:             mid = x;
10:        }
11:        */
12:    }
13:    else {
14:        if (x > y) {
15:            mid = y;
16:        }
17:        else if (x > z) {
18:            mid = x;
19:        }
20:    }
21:    return mid;
22: }

```

Figure 6.1.: Faulty `mid()` Method. Each Syntactic Block is Enclosed by a Box

table, each TC corresponds to a column, with the top of the column corresponding to the inputs, the black dots corresponding to coverage, and the status of the test at the bottom of the column. To the right of the test case columns are the Ochiai score (as defined in chapter 2) and the corresponding rank for each element. Furthermore, if an element localizes the fault, the corresponding row is highlighted.

As mentioned previously, it is not possible to localize a fault of omission using the statement granularity. Therefore, no SBFL technique using the statement granularity will be able to localize the fault in the `mid()` method.

Table 6.1.: *mid()* Statement Granularity Ochiai Calculations

Line #	Test Cases						Ochiai	Rank
	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1	•	•	•	•	•	•	0.577	2
2	•	•	•	•	•	•	0.577	2
3	•	•	•	•	•	•	0.577	2
4	•	•			•	•	0.707	1
5		•					0	6
13			•	•			0	6
14			•	•			0	6
15			•				0	6
17				•			0	6
18							0	6
21	•	•	•	•	•	•	0.577	2
Verdict	F	P	P	P	P	F		

Table 6.2.: *mid()* Method Granularity Ochiai Calculations

Line #s	Test Cases						Ochiai	Rank
	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1-22	•	•	•	•	•	•	0.577	1
Status	F	P	P	P	P	F		

To localize the fault using the method granularity, the only information provided to the developer is that the fault is contained in the *mid()* method (see Table 6.2). However, due to the unit test nature of the test cases, this fact is obvious. A developer would still have to go through all the statements in the method to find the fault.

As seen in Table 6.3, when using the syntactic block granularity, the fault is localized to the then block of the if statement (*ITB* block) from lines 3-12 with a rank of 1. As an improvement over the method granularity result, the developer would only have to look through one portion of the method to find the fault. Furthermore, the developer would have a further intuition as to the location/type of fault that exists. Due to the average depth of faults within the program elements of the syntactic block granularity, the developer would expect the fault to likely be an issue with the direct children elements of the block, or the block itself. In this case, the faulty missing element would in fact be a direct child of the top ranked *ITB* block from lines 3-12. Furthermore, due to the 0 suspiciousness score of the *ITB* block from lines 4-6, the developer can infer that the fault is either with the if statement conditional at line 4, or is a fault of omission.

This added information provided by the syntactic block granularity, as well as the reduced

Table 6.3.: *mid()* Syntactic Block Granularity Ochiai Calculations

Block Type	Line #'s	Test Cases						Ochiai	Rank
		3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
MD	1-22	•	•	•	•	•	•	0.577	3
ICS	3	•	•	•	•	•	•	0.577	3
ITB	3-12	•	•			•	•	0.707	1
ICS	4	•	•			•	•	0.707	1
ITB	4-6		•					0	5
IEB	13-20			•	•			0	5
ICS	14				•	•		0	5
ITB	14-16			•				0	5
ICS	17					•		0	5
ITB	17-19							0	5
Verdict		F	P	P	P	P	F		

number of statements required to search through to localize the fault, helps illustrate the benefits of the syntactic block granularity over the other two granularities.

6.3. Syntactic Block Granularity

Each syntactic block consists of a set of statements that syntactically belong together to form a program element. For instance, every statement in a method declaration belongs together, as whenever the method is called, the contained statements may be run. Furthermore, each element may further be broken into multiple sub-elements. For example, an if statement has three components: the *condition statement*, the *then block*, and the *else block*. Each of these sub-elements may be run separately from each other. For instance, the *condition statement* will always be executed, but depending on the boolean value of the conditional, either the *then block* or the *else block* will be executed. As a result, for Java programs, the syntactic block granularity consists of the 18 different types of program elements found in Table 6.4. For ease of use, each syntactic block type has an ID associated with it. An example of each type of syntactic block can be found in bold in the last column in Table 6.4.

Like the method granularity, for each syntactic block, if any of the contained statements are executed, the syntactic block is also marked as executed. Originally, *Class*, *Interface*, and *Enum* declarations were also considered as types of syntactic blocks. However, due to their average size compared to all other types of blocks, the added benefit of encompassing class level faults (such as missing method declarations or incorrect class variables) was outweighed by the overall added wasted effort associated with inspecting whole classes for a fault.

Due to the hierarchical nature of syntactic blocks, it is possible for one block to completely

Table 6.4.: Java Syntactic Block Types

Base Program Element	Syntactic Block Type	Type ID	Example
-	Constructor declaration	CND	<code>class X { X() { ... } }</code>
-	Method declaration	MD	<code>public int foo() { ... }</code>
-	Initializer declaration	IND	<code>static { a = 3; }</code>
-	Do statement	DS	<code>do { ... } while (a == 0);</code>
Foreach statement	Condition statement	FECS	<code>for (Object o : objects) { foo(); }</code>
	Body block	FEBB	<code>for (Object o : objects) { foo(); }</code>
For statement	Condition statement	FCS	<code>for (int a = 3; a <10; a++) { foo(); }</code>
	Body block	FBB	<code>for (int a = 3; a <10; a++) { foo(); }</code>
If statement	Condition statement	ICS	<code>if (a == 5) foo() else bar();</code>
	Then block	ITB	<code>if (a == 5) foo() else bar();</code>
	Else block	IEB	<code>if (a == 5) foo() else bar();</code>
Switch statement	Statement	SS	<code>switch(a) { ... }</code>
	Entry statement	SES	<code>case 1: foo(); break;</code>
Try-Catch statement	Try block	TCTB	<code>try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }</code>
	Catch block	TCCB	<code>try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }</code>
	Finally block	TCFB	<code>try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }</code>
While statement	Condition statement	WCS	<code>while (a > 0) { bar(); }</code>
	Body block	WBB	<code>while (a > 0) { bar(); }</code>

encapsulate another. For example, in Figure 6.1 the *ITB* block from lines 3-12 encapsulates the *ICS* block at line 4 and the *ITB* block from lines 4-6. Because of this, sections of code will appear multiple times as a programmer walks through the ranked list of elements. In order to prevent unnecessary work, any block completely encapsulated by another block with a higher suspiciousness score can be ignored and removed from the final ranking.

While our work focused on faults in Java programs, the concept would be similar for other programming languages with similar syntax, e.g., C, C++, C#, Go, PHP, and Swift.

6.4. Experiments

In the following section, we will define implementation details and research questions.

6.4.1. Data set

As mentioned earlier, due to the nature of the faults in the Defects4J data set, not all bugs can be localized by each granularity. For instance, the statement granularity can only localize 228 bugs, the method granularity 386 bugs, and the syntactic block granularity 388. There are 225 bugs that overlap across all three granularities¹. For the sake of comparison, all three granularities will be compared on these 225 bugs. This data set does not contain any faults of omission and will be referred to as the **no-omission data set**. Furthermore, the method and syntactic block granularities will be compared on the 386 bugs that can be localized by the method granularity, which includes the faults of omission, and will be referred to as the **full data set**. Due to them being class level faults, the following bugs are unable to be localized by any of the three granularities: Chart-23, Closure-28, Lang-23, Lang-56, Math-12, and Math-104. They will be ignored for this evaluation.

6.4.2. Implementation

The following outlines our implementation details and is broken down into three parts: spectra generation, preprocessing, and SBFL ranking metrics.

Spectra Generation

GZoltar Statement Spectra Generation. The process for generating the method and syntactic block granularity spectra files relies on having access to the statement granularity spectra. By running the GZoltar command-line tool (v1.6.2) [151] on all 395 bugs in the Defects4J database, we constructed the statement granularity spectra using the same data generation process as utilized by Pearson et al. for generating GZoltar statement spectra in Defects4J [142]. This process results in **matrix** and **spectra** files, where each line in the matrix file corresponds to a test case, with each line consisting of columns of 1s and 0s corresponding to whether a program element is executed (1) or not (0) in the given test case, as well as an indicator if the test is passing (+) or failing (-). Each line in the produced spectra file is the label of a program element and corresponds to a given column in the matrix file.

Method and Syntactic Block Spectra Generation. After constructing all statement granularity GZoltar files, we utilize the JavaParser library² to walk through the abstract syntax tree of the source code files which are loaded during the execution of the failing tests³. We then construct new program elements based on the 18 types of syntactic blocks found in

¹The three bugs that cannot be localized by every granularity are Lang-25, Mockito-26, and Time-11. Lang-25 is a fault in a static class variable, which only the statement granularity can localize. The other two are faults in a static initializer declaration, which can be localized by both the statement and syntactic block granularities, but not the method.

²<http://javaparser.org/>

³provided by Defects4J framework under `defects4j/framework/projects/[project]/loaded_classes/[bug].src`

Table 6.4. The corresponding GZoltar matrix file is then constructed by walking through the statement granularity GZoltar matrix file. For each syntactic block element that encapsulates an executed statement for a given test, the corresponding matrix cell is set to 1. All syntactic block elements that do not encapsulate an executed statement are otherwise set to 0.

The same process is repeated for constructing the method granularity files, except it only considers *CND* and *MD* blocks.

Preprocessing

Spectra Update using Dynamic Call Graph. During the development of our system, it was discovered that the GZoltar matrix files would sometimes be missing the execution information for the faulty program element, even if it was executed during the failing test. This was discovered by inspecting the dynamic call graphs generated by the TestCaseCallGraph tool⁴. In some bugs, for example Chart-16, the faulty element was contained in a method that appeared in the dynamic call graph for the failing test cases, but the corresponding statements were set to 0 in the GZoltar matrix file. Through inspecting the stack traces for the failing test cases by hand, we determined that this was likely due to an exception being raised in the first executed statement of a given method. As a result, we verified that the method's presence in the dynamic call graph was correct, whereas the missing execution of the statement in the GZoltar spectra files was not.

In order to make up for this missing execution data, the GZoltar matrix file was updated by setting the elements corresponding to methods appearing in the dynamic call graph for a given failing test to 1, even if it was set to 0 in the original matrix. In all, 5 bugs in Defects4J were updated in this manner⁵. While this is not a perfect solution, as we were only able to update *MD* and *CND* blocks, it did help improve the overall results.

Spectra Dimensionality Reduction. The final data preparation step consists of reducing the dimensionality of the GZoltar matrix file. For this step, we make the assumption that a faulty program element needs to be executed in order to manifest as a failure. Based on this assumption, all program elements which were never executed in any tests (i.e. all columns containing only 0s) were removed from the GZoltar matrix and spectra files. Furthermore, after reducing the number of considered program elements, all tests which do not contain any executed program element (i.e. all rows containing only 0s) were also removed from the GZoltar matrix file.

The reasoning behind these actions were twofold. First, it is assumed that the fault localization performance would improve when considering fewer program elements and tests, as the presence of non-executed program elements would only add to the wasted effort, and tests covering none of the considered program elements do not add any extra information. Second, the reduced dimensionality leads to better time and memory performance for the SBFL ranking step.

⁴<https://github.com/dkarv/jdcallgraph>

⁵Chart-16, Closure-99, -100 and -103, and Math-89

SBFL Suspiciousness Metrics

For evaluation, we consider the performance on three of the more popular SBFL metrics mentioned in Chapter 2 namely DStar2, Ochiai, and Tarantula.

6.4.3. Research Questions

In order to evaluate our contribution, we define the following research questions.

- **RQ1:** How does the syntactic block granularity compare to other granularities? In order to determine the effectiveness of our proposed syntactic block granularity, we look to compare it with the commonly used statement and method granularities. To ensure similarity with the syntactic block data, both the statement and method granularity spectra files undergo the same preprocessing steps as the syntactic block granularity files. Namely, we run the spectra update step for the method granularity, and the dimensionality reduction step for both the method and statement granularities. We evaluate each granularity using the standard SBFL ranking metrics of DStar2, Ochiai, and Tarantula. Furthermore, all three granularities are compared using the no-omission data set, whereas just the method and syntactic block granularities are compared using the full data set.
- **RQ2:** How does the locality of program elements to the faults compare among the three granularities? When inspecting program elements containing multiple statements, it is important to be able to have some insight as to which statements are more suspicious than others. For example, when inspecting a method for a fault, it would be helpful for the user to know where to start looking for the fault, instead of having to walk through each statement one-by-one. By knowing certain characteristics of the program elements in each granularity, a user may be able to localize the fault easier. One possible characteristic to consider would be the proximity of the most suspicious faulty program element e^* to the fault itself. We hypothesize that if the average depth of the fault in the abstract syntax tree of the program element e^* is low, it will be easier to find the fault, as the user could work their way down the AST, giving higher priority to the shallower elements.

To evaluate the proximity of the e^* to the fault itself, we introduce a new metric, called **fault locality depth**, or **FLD**, which calculates the depth of the fault within the abstract syntax tree of the program element e^* . For example, consider the fault-localizing syntactic blocks in Figure 6.1. The depth of the fault in the AST of the *MD* block would be 1, as there is one program element that is closer to the fault's location, whereas the depth of the faulty element in the AST of the *ITB* block from lines 3-12 would be 0, as it is the faulty element itself.

FLD refers to how “deep” a fault is within a given element. The depth is calculated with respect to the given element's AST. What we convey with FLD is that it gives

developers a better idea as to which lines to check first. Going back to the example, when a developer looks into ITB 3-12, thanks to this FLD characteristic, she intuitively knows that lines 4 and 5 are less likely to contain the fault, because both of those lines correspond to a “child” program element of ITB 3-12. So she has more intuition as to where the fault has likely occurred (i.e., she may look for the missing code block before looking into lines 4-6 for a fault).

Due to the nature of the statement granularity, it is obvious that the fault locality depth of every fault-localizing program element would be 0, as the statement is the faulty element itself. Therefore, for this evaluation, we only consider the method and syntactic block granularities. To stay true to the other evaluation, we evaluate the two granularities on both the no-omission and full data sets.

6.5. Results

In the following section, we will describe the evaluation results and discuss threats to validity. **Note on syntactic block evaluation:** As mentioned earlier, it is possible for one block to completely encapsulate another. Therefore, to prevent double-counting when calculating the absolute rank and EXAM scores, we ignore any blocks that have already been covered by blocks of a strictly higher suspiciousness score. Furthermore, when counting the wasted effort, we only count the first time a statement shows up in the ranking, regardless of it being first in the super-block or sub-block.

6.5.1. Evaluation Results

RQ1:The results from running the three considered SBFL metrics for each of the three granularities on the no-omission and full data sets are displayed in Tables 6.5 and 6.6 respectively. Furthermore, additional metrics for this specific evaluation can be found in Tables 6.7 and 6.8 respectively. The top performing value for each granularity for each evaluation metric is in bold, with the top performing value overall highlighted in gray.

When considering the no-omission data set results in Table 6.5, the statement and syntactic block granularities have very similar EXAM scores, with syntactic block having slightly better scores overall, except for the case of average O-EXAM. This means that the syntactic block granularity exhibits a similar, if not better, proportional wasted effort to the statement granularity. As expected, the method granularity EXAM scores are noticeably worse than the other two, with the one exception being Δ -EXAM. This is due to the larger average size of the elements examined with the method granularity. Furthermore, the smaller Δ -EXAM score can be attributed to the smaller number of ties seen with the method granularity due to the reduced number of elements being considered.

When comparing the best and worst absolute rankings, the statement granularity does the worst overall, exhibiting the worst values over all granularities in all but the best

Table 6.5.: RQ1 - No-omission Data Set Results

Granularity	Metric	EXAM						Best						Worst					
		O-Avg	O-Med	P-Avg	P-Med	Δ -Avg	Δ -Med	T1	T5	Top-10	Avg	Med	T1	T5	Top-10	Avg	Med		
Statement	DStar2	0.471	0.031	0.735	0.088	0.264	0.016	68	103	118	251.78	8	13	50	78	352.19	31		
	Ochiai	0.477	0.037	0.742	0.091	0.264	0.016	67	101	117	252.63	8	13	49	78	353.02	30		
	Tarantula	0.499	0.026	0.764	0.090	0.265	0.018	67	102	117	254.94	8	10	46	76	355.43	31		
Method	DStar2	0.758	0.056	0.991	0.107	0.233	0.006	65	118	141	43.73	5	38	95	124	51.77	8		
	Ochiai	0.766	0.062	0.999	0.127	0.234	0.006	64	116	140	43.92	5	38	93	124	51.97	9		
	Tarantula	0.829	0.069	1.062	0.132	0.233	0.006	66	115	134	45.85	5	36	93	119	53.86	9		
Syntactic Block	DStar2	0.501	0.016	0.724	0.072	0.223	0.012	81	122	140	81.42	4	28	76	103	100.81	13		
	Ochiai	0.505	0.018	0.727	0.077	0.222	0.012	80	121	136	82.03	4	27	75	103	101.37	14		
	Tarantula	0.537	0.018	0.761	0.084	0.223	0.012	79	121	138	85.86	4	22	72	99	105.67	14		

Table 6.6.: RQ1 - Full Data Set Results

Granularity	Metric	EXAM						Best						Worst					
		O-Avg	O-Med	P-Avg	P-Med	Δ -Avg	Δ -Med	T1	T5	Top-10	Avg	Med	T1	T5	Top-10	Avg	Med		
Method	DStar2	0.733	0.055	0.907	0.105	0.174	0.006	117	202	236	45.78	5	75	167	213	52.60	8		
	Ochiai	0.736	0.056	0.911	0.115	0.175	0.006	118	202	236	45.92	5	75	167	213	52.75	8		
	Tarantula	0.806	0.064	0.980	0.126	0.175	0.006	117	198	230	49.47	5	69	164	208	56.29	8.5		
Syntactic Block	DStar2	0.480	0.026	0.642	0.092	0.162	0.012	127	196	224	81.47	5	43	124	169	96.57	15		
	Ochiai	0.481	0.027	0.643	0.092	0.162	0.012	127	194	222	81.92	5	42	122	169	96.98	15.5		
	Tarantula	0.527	0.030	0.690	0.099	0.163	0.013	124	192	225	89.97	6	36	116	164	105.46	16		

Table 6.7.: RQ1 - No-omission Data Set Additional Metrics

Granularity	Metric	Average Size	Median Size
Method	DStar2	27.151	14
	Ochiai	27.151	14
	Tarantula	26.858	14
Syntactic Block	DStar2	5.831	2
	Ochiai	5.831	2
	Tarantula	5.622	2

Table 6.8.: RQ1 - Full Data Set Additional Metrics

Granularity	Metric	Average Size	Median Size
Method	DStar2	25.443	14
	Ochiai	25.443	14
	Tarantula	25.295	14
Syntactic Block	DStar2	7.236	3
	Ochiai	7.236	3
	Tarantula	7.168	2.5

absolute rank Top-1 count. This means that for the statement granularity, the program element e^* will often be further down the list of suspicious elements when compared to the other two granularities. This can partially be attributed to the larger amount of ties and overall number of elements considered with the statement granularity. When comparing the other two granularities, method and syntactic block perform similarly in the best-case, with syntactic block having the edge in best absolute rank Top-1 and Top-5 counts, but method having the edge in the average best absolute rank. The difference in averages can again be attributed to the difference in element counts, as the method granularity has to consider less elements overall. However, when considering the median best absolute rank, the syntactic block granularity actually has the edge. That being said, when considering the worst-case absolute rank metrics, the method granularity performs the best in all categories, which again can be attributed to the number of elements considered.

The results from the full data set found in Table 6.6 are very similar to those from the no-omission data set. The syntactic block granularity exhibits a clear advantage over the method granularity in the EXAM scores, whereas both have a similar performance in the best-case absolute rank values. Finally, the method granularity performs better in the worst-case absolute rank values.

While the method granularity shows an obvious advantage over the other two granularities when it comes to the absolute rank metrics, especially in the worst-case, this may be due to the overall bias of the absolute rank metrics towards granularities with fewer elements to examine. Since there are fewer elements to examine, there are likely fewer ties, and the probability that an element will be in the Top-N elements is higher. Furthermore,

Table 6.9.: RQ2 - No-omission Data Set Depth Metrics

Granularity	Metric	Average FLD	# Elements with FLD				
			0	1	2	3	4+
Method	DStar2	0.929	116	49	36	14	10
	Ochiai	0.929	116	49	36	14	10
	Tarantula	0.920	117	49	35	14	10
Syntactic Block	DStar2	0.098	205	19	0	1	0
	Ochiai	0.098	205	19	0	1	0
	Tarantula	0.093	206	18	0	1	0

the absolute rank metrics ignore the fact that the size of the elements are different. The effort of examining two methods is clearly not better than examining four statements or possibly even four syntactic blocks.

This issue can be further illustrated by the average and median size metrics found in Tables 6.7 and 6.8. These values are based on the number of statements contained in the program element e^* for each bug in the data sets. Since it is obvious that the statement granularity would have the best size metrics (1 for both average and median), only the method and syntactic block granularities were included in these tables. For the no-omission data set (as seen in Table 6.7) the syntactic block granularity is not far behind the statement granularity, with an average size of 5.622 statements and median of just 2. On the other hand, the method granularity has an average size of 26.858 statements and median of 14. This points to the syntactic block granularity being a lot closer to the statement granularity in terms of effort to examine each element. Again, for the full data set (as seen in Table 6.8), the values follow a similar pattern, with the syntactic block granularity exhibiting a 7.168 average and 2.5 median versus method's 25.295 and 14 respectively.

The final difference between the three granularities can be seen in the number of bugs in the Defects4J data set that they can localize. Due to its fine-grained nature, the statement granularity can only localize 228 bugs, or only 57.72% of the 395 bugs in the data set. Due to its more coarse-grained nature, the method granularity can localize all but 9 bugs, or 97.72% of the bugs in the data set. Finally, the syntactic block granularity can localize all 386 bugs that method can localize, as well as 2 additional bugs which contain faults within a static initializer declaration. This gives the syntactic block granularity a 98.23% coverage of the bugs in the Defects4J data set.

As a recap, the syntactic block granularity exhibits EXAM scores that are similar, if not better than those of the statement granularity (an improvement over the method granularity), while having best absolute rank values similar to the method granularity (an improvement over the statement granularity). Furthermore, it is able to localize more faults (388) than both the statement (228) and method (386) granularities.

RQ2: The **fault locality depth** metrics for the no-omission and full data sets can be found in Tables 6.9 and 6.10 respectively. As in **RQ1**, the top-performing value for each granularity

Table 6.10.: RQ2 - Full Data Set Depth Metrics

Granularity	Metric	Average FLD	# Elements with FLD				
			0	1	2	3	4+
Method	DStar2	0.902	199	91	55	25	16
	Ochiai	0.902	199	91	55	25	16
	Tarantula	0.891	201	91	53	25	16
Syntactic Block	DStar2	0.070	361	24	0	1	0
	Ochiai	0.070	361	24	0	1	0
	Tarantula	0.067	362	23	0	1	0

for each evaluation metric is in bold, with the top-performing value overall highlighted in gray.

As mentioned previously, for the statement granularity, it is obvious that the fault locality depth of every fault-localizing program element would be 0, as the statement is always the faulty element itself. So for this evaluation, we only consider the method and syntactic block granularities.

When comparing the results of the method and syntactic block granularities on the no-omission data set (as seen in Table 6.9), we found that the best average FLD was 0.92 for the method granularity, and 0.093 for the syntactic block granularity. Similarly, the results on the full data set (as seen in Table 6.10) were 0.891 for the method granularity and 0.067 for the syntactic block granularity, resulting in up to a 92.48% improvement (percent decrease) in average FLD. For the syntactic block granularity, this means that on average, the faulty program element e^* was more often than not, the fault itself. In fact, when looking at the FLD values for the Tarantula ranking metric on the full data set, only 24 e^* elements were not the closest block to the faulty element, with 23 of them being the second closest, and 1 (Math-81) being the fourth closest. In comparison, 185 of the e^* elements for the method granularity were not the closest block, with 91, 53, and 25 of them being the second, third, and fourth closest respectively. This difference exhibits one of the key benefits of the syntactic block granularity. Due to the locality of the most suspicious, fault localizing blocks to the faults themselves, a user often has a hint as to the location of the fault.

For example, consider the faulty version of the `mid()` method in Figure 6.2, where the fault is now on line 19. Thanks to the average fault locality depth of the syntactic block granularity, when inspecting the `MD` block from lines 1-20, a developer would have some intuition as to which lines to focus on first. Instead of giving every statement in `mid()` the same priority, as would be the case when using the method granularity, a developer could first focus on lines 1, 2, and 19, as they all have a depth of 0 in the AST of the `MD` block. Using this intuition, the developer would be able to skip over 75% of the statements when first looking for the location of the fault. In this case, this would allow the user to localize the fault much faster than normal. Therefore, this characteristic provides the user with additional information as to where to start looking for the fault inside a program element.

```

1:  public int mid(int x, int y, int z) {
2:      int mid = z;
3:      if (y < z) {
4:          if (x < y) {
5:              mid = y;
6:          }
7:          else if (x < z) {
8:              mid = x;
9:          }
10:     }
11:     else {
12:         if (x > y) {
13:             mid = y;
14:         }
15:         else if (x > z) {
16:             mid = x;
17:         }
18:     }
19:     return z; // faulty element
20: }

```

Figure 6.2.: Faulty `mid()` Method, with Fault on Line 19. Each Syntactic Block is Enclosed by a Box

While we made the assumption in our experiments of perfect bug detection, this is often not the case in practice [182, 135, 142]. The normal user will not be able to perfectly locate bugs every time if just given an element containing a fault, but if given the right information and insights, their chances will increase. Therefore, we would argue that this added information helps differentiate the syntactic block granularity from the method granularity, due to the insight each block gives as to the possible location of a contained fault.

As a recap, while the statement granularity is still the best, the syntactic block granularity has a definite advantage over the method granularity in its locality to the faulty statements, exhibiting up to a 92.48% improvement in average fault locality depth. This characteristic

helps provide insight into the possible location of the contained fault, an advantage that the method granularity does not exhibit.

6.5.2. Threats to Validity

Our results showed the syntactic block granularity to be an improvement over other granularities in our context. Nevertheless, we do not claim that our experiment setup and results will be valid for all other contexts.

We ran our evaluations on the bugs in the Defects4J library, and while we do not claim that our results can be generalized to bugs in the field, Defects4J is the largest available database of real bugs for Java and it includes bugs from 6 different open-source projects. However, due to the data set still being relatively small in size, it is possible that our findings may be restricted to Defects4J. Additionally, the quality and coverage of the test suites considered in our evaluations may not reflect the quality of those in the real world. Defects4J includes projects with relatively good test coverage, so when considering another data set with a different quality of tests, the results using our technique may not be similar to what we have produced. Furthermore, one drawback to using the bugs in Defects4J is that each program version only contains a single isolated bug. While a majority of the research into fault localization focuses on programs with a single bug, this is not always the case for real-life software [182].

Additionally, during evaluation, we made the simplifying assumption of perfect bug detection. As mentioned previously, this is often not the case in practice [182, 135, 142]. However, it allows for collecting some objective information on the effectiveness of a given granularity and provides a common ground for comparing all three granularities [135].

Finally, a few steps in our implementation were predicated on the assumption that a faulty element must be executed in order to be marked as suspicious. In other words, we did not consider program elements which were not executed in at least one failing test case as possible locations for a fault. While this is one of the main underlying assumptions of spectrum-based techniques in general, it is still possible for elements to be faulty, even if they are not executed in a failing test case. Mockito-9 is the one instance of such a fault in Defects4J. However, this fault is an issue of incorrect dependencies, rather than faulty logic or implementations, and is not a good use case for any SBFL technique.

6.6. Conclusion

In this chapter, we introduced the syntactic block granularity, a new program spectra granularity for the use in SBFL techniques. It considers a wider range of program elements than any other program spectra granularity, while providing additional insight into the possible location of the fault within the given program element.

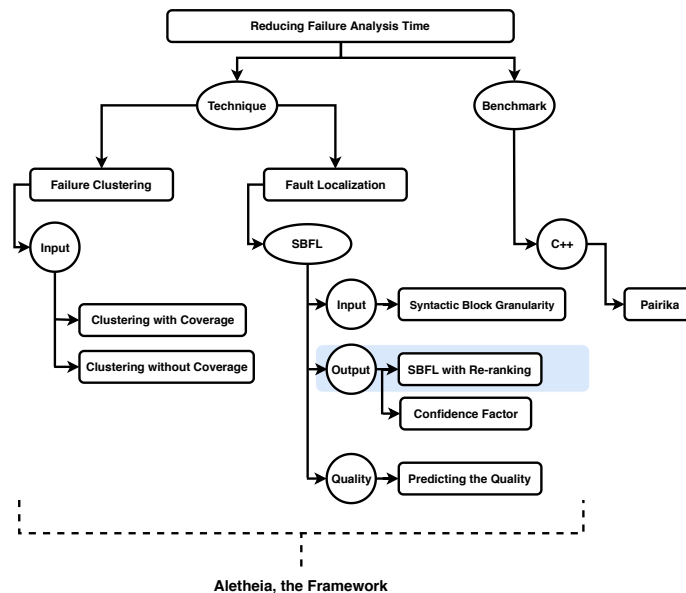
Considering a set of commonly used evaluation metrics, we evaluate our new syntactic block granularity on Defects4J. We found that our proposed syntactic block granularity

exhibits best-case absolute ranking behavior similar to the method granularity, while having a wasted effort equivalent to, if not better than, the statement granularity. Furthermore, it covers more types of faults than both existing granularities. Finally, when compared to the method granularity, it exhibits up to a 92.48% improvement when it comes to the locality of the program elements to the fault, a characteristic that provides the user with a better insight into the possible location of the faults.

While the improvement of the syntactic block granularity is a step towards addressing the issues of SBFL, it is still uncertain if it makes SBFL usable in the real world. For instance, an O-EXAM score of even 0.48 may still be too high, especially when considering very large, real-world programs. Our contribution is an improvement of the state-of-the-art, however it is likely not the final solution to making SBFL applicable in practice.

7. Improving SBFL Through Tackling Confounding Bias

This chapter presents a new ranking technique to improve the effectiveness of spectrum-based fault localization.



In another attempt to improve the effectiveness of SBFL techniques, we propose a new re-ranking approach that takes into account dynamic call and explicit data-dependency graphs of failing tests. By evaluating our technique on Defects4J bugs, we show that our re-ranking approach outperforms the popular SBFL techniques, namely DStar, Ochiai, Tarantula, and causal inference-based techniques, namely MFL.

7.1. Introduction

As previously described in Chapter 2, SBFL techniques measure the correlation between program elements and test failures to compute suspiciousness scores. However, one **problem** is that these techniques do not control potential confounding bias.

```
public void F1(int i) {  
    if (i < 0) {  
        ... // Faulty  
        F2(x);  
    } else {  
        ...  
        F3(x);  
    }  
}
```

Figure 7.1.: A Hypothetical Faulty Method with Two Branches

Figure 7.1 exhibits an example of confounding bias in SBFL results. Assume that a fault in method F1 propagates only through the first branch where method F2 is triggered, while the second branch where methods F3 is called, executes correctly. Put differently, although F1 contains a fault, only those tests taking the left branch are failing. In this case, an SBFL technique gives the highest suspiciousness score to the method F2, since it's executed more frequently in failing executions and less frequently in passing executions (F1: 1 failing, 1 passing, F2: 1 failing, and F3: 1 passing). However, method F1 is the faulty element.

“Confounding bias happens when a seeming causal impact of an event on a failure may be in fact due to another unknown confounding variable, which causes both the event and the failure” [119]. Given a program and a test suite, assume that, all failing TCs induce dependence chain (chain of executed program elements in our case) $e_1 \rightarrow e_2 \rightarrow e_{bug} \rightarrow e_3 \rightarrow e_4 \rightarrow e_{fail}$ and all passing TCs induce $e_1 \rightarrow e_2$ only, where e_{bug} indicates the execution of faulty element and e_{fail} indicates a failure. A correlation-based approach such as SBFL would assign the same suspiciousness score to any of e_{bug} , e_3 , or e_4 , thus resulting in two false positives, whereas a causation-based approach that considers dependencies to have causal effect would assign e_4 the lowest suspiciousness score and e_{bug} the highest suspiciousness score. This means, when computing the suspiciousness scores, the confounding bias to consider for e_4 would involve e_3 and e_{bug} , for e_3 it would involve e_{bug} , and no confounding is involved when computing the suspiciousness score of e_{bug} [119].

Shu et al. [162] introduced MFL, a method-level FL technique based on causal inference to tackle this issue (see Chapter 2). Since we could not find any implementation of this approach or any evaluation of it in survey papers, we implemented, improved, and evaluated it on Defects4J data set. Our results show that some SBFL techniques such as DStar [185] outperform MFL.

To the best of our knowledge, MFL and its ancestors [8] are the only works attempting to reduce confounding bias in FL. In our further analysis of SBFL results, we observed that often if the most suspicious element s^* does not contain a fault, one of its Markov blanket [137] members, namely its parents, its children, and its children's other parents, contains a fault, as shown in Figure 7.1. Method F2 is the most suspicious element, while its parent F1 contains a fault. To improve SBFL effectiveness, we propose a re-ranking strategy based on this observation.

As a **solution**, we augment SBFL with a combined dynamic call and data-dependency graphs of failing tests. Since DStar is shown to be one of the best performing SBFL techniques in different studies such as [182], and method-level to be the mostly used granularity level [94], we use method-level DStar as the base in our approach¹. First, using DStar, we find the most suspicious method s^* of the program. It has rank 1 on the suspiciousness ranking list. We locate it on the combined dynamic call graph of the failing tests and inspect it. If it is faulty, the process ends. If it is not faulty, we list all of its parents and grandparents. Instead of inspecting the method that is given rank 2 on the DStar suspiciousness ranking list, we inject parents and grandparents of s^* between rank 1 and rank 2 and re-rank all methods accordingly. The re-ranking approach gives the second rank to the parents of s^* and the third to its grandparents. We continue inspecting based on the new ranking until we find the fault. Visual representation of call graph while highlighting the most suspicious elements on it aids users in better understanding the problem.

Our results show that with our re-ranking approach, we are able to improve the effectiveness of SBFL, however slightly. The results raise a question, whether just improving the SBFL effectiveness is enough to bridge the gap between research and industry? The honest answer to this question is no. Although we are able to improve the effectiveness and provide a visual guide via highlighting the suspicious parts of the method call structure, there is no guarantee that the SBFL results are always helpful. In fact, no matter what is being used as the FL tool, it is not clear whether the results are helpful or not. To help users in better deciding whether to consider the SBFL ranking list or not, we need a *Confidence Factor* (see Chapter 10). This factor informs users about the probability of the fault being in the Top-10 positions.

In addition, we believe that it is time to shift the focus from merely improving the automated FL techniques to the quality of the code to facilitate fault localization. As the starting point, we looked deeper into the source code of some of the studied buggy versions where automated FL is not so effective. Using some examples, we show that some programming styles can have a negative impact on the effectiveness of FL. Chapter 8 provides a deeper look into this issue.

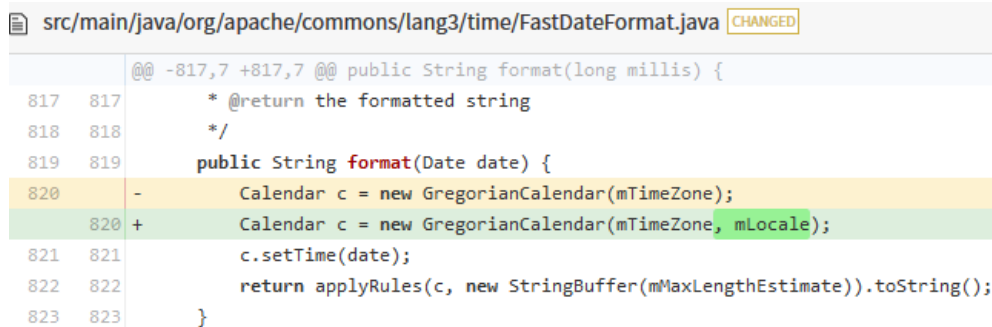
In this chapter:

- We propose a re-ranking approach for SBFL techniques which leverages dynamic call and data-dependency graphs of failing executions. This approach produces more accurate results augmented with the graphical representation of suspicious elements which can enhance users' understanding of the program's faulty behavior. Utilizing this technique, programmers find out the direction they should search for the fault.
- We propose some practical suggestions on how to improve the code to facilitate fault localization.

¹We introduce syntactic block granularity in Chapter 6 as a better option than method granularity. However, for the sake of comparison with other studies especially the MFL technique, we use method granularity in this experiment.

7.2. Methodology

In the following, we outline our approach. We use a real bug, Lang-26, from Defects4J as our motivating example. This bug is a wrong method reference that causes one test to fail [163].



```

src/main/java/org/apache/commons/lang3/time/FastDateFormat.java CHANGED
@@ -817,7 +817,7 @@ public String format(long millis) {
817 817     * @return the formatted string
818 818     */
819 819     public String format(Date date) {
820 820 -     Calendar c = new GregorianCalendar(mTimeZone);
820 820 +     Calendar c = new GregorianCalendar(mTimeZone, mLocale);
821 821     c.setTime(date);
822 822     return applyRules(c, new StringBuffer(mMaxLengthEstimate)).toString();
823 823 }

```

Figure 7.2.: Human Patch to Fix Lang-26

7.2.1. Approach

Step 1: Finding the most suspicious method(s) using SBFL

As mentioned earlier, we utilize DStar2 to find the most suspicious method in the first step. DStar2 calculations on Lang-26 spectrum places “lang3.time.FastDateFormat-TextField-1171” and “lang3.time.FastDateFormat-StringLiteral-1130” methods at rank 1, as the most suspicious methods. Method “lang3.time.FastDateFormat-820” which is the faulty method gets rank 17.

Step2: Locating the most suspicious method(s) on the dynamic call and/or data-dependency graph

In the second step, we generate a graph which includes dynamic method calls and/or explicit data-dependencies of all failing tests.

A dynamic call graph is generated at runtime by monitoring the program execution. The graph contains nodes that indicate the executed methods and edges between methods that represent method calls. We consider dynamic call graph to inspect real, not potential (as is the case in static call graphs) dependencies.

An explicit data-dependency graph indicates dependencies between program elements introduced by a common variable used in multiple program elements. A data-dependency exists when two program elements exchange data using a variable. This happens when one program element writes to a field, and another element reads that field later. The result is a

data-dependency between the first and second elements. Figure 7.3 shows an example of data-dependency between methods.

```

void compute(int x) {
    this.result = x * 10;
}
int getResult() {
    return this.result;
}
int f() {
    compute(5);
    return getResult();
}

```

Figure 7.3.: An Example of Data-dependency Between Methods

Considering Figure 7.4, a call graph contains only the solid lines. A data-dependency graph contains only the dashed line. A combined graph can be helpful in FL. If method *compute* is faulty, method *getResult* will also return a wrong result. Thus, it will be labeled as a suspect. Looking into the combined graph, one can improve the labeling by adjusting for the confounder.

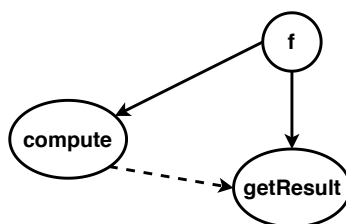


Figure 7.4.: Combined Call/Data-dependency Graph of Figure 7.3

Figure 7.5 shows the call graph of Lang-26 failing test. Due to space constraints, it is only depicting the left branch. Thick red boundaries highlight the most suspicious methods. All nodes are annotated with their ranks. In this example, we use the best ranks in case of ties. At this step, the user receives the graph and inspects these two methods. Since the fault is not there, the user continues the process through step 3.

Step3: Re-ranking program methods

In step 3, we find the parents and grandparents of the DStar's most suspicious methods. Then, we inject them between rank 1 and rank 2 and change all the ranks accordingly. Parents get rank 2 and grandparents get rank 3. In our example, "lang3.time.FastDateFormat-888" is the parent and "lang3.time.FastDateFormat-820" is the grandparent. Thus, in our new list, we place them right after ranked 1 methods and change their ranks to

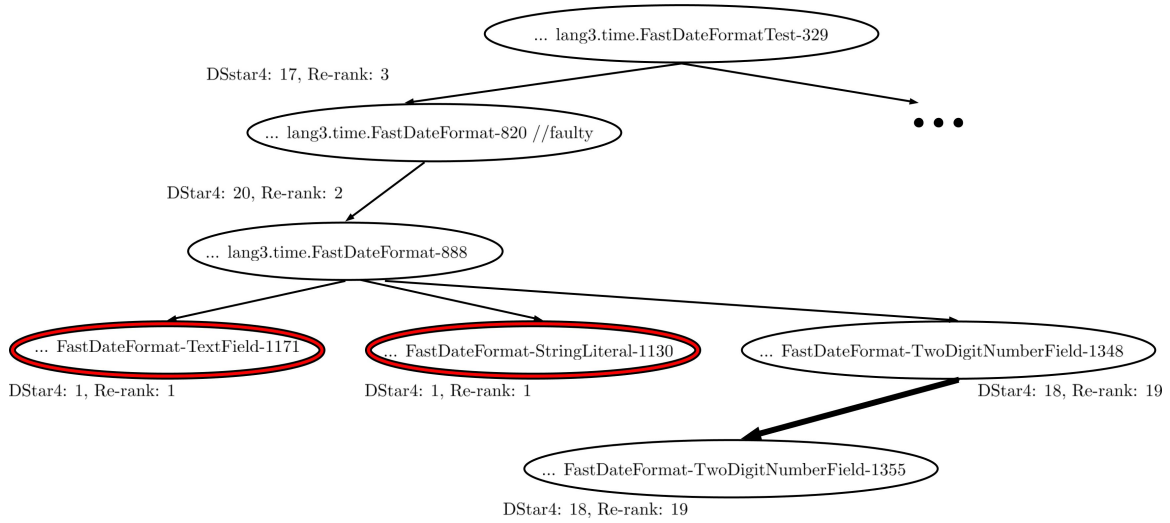


Figure 7.5.: Left Branch of Lang-26 Call Graph. Numbers in the Nodes Indicate the Line Numbers of Methods in the Source Code.

2 and 3 respectively. As annotations on Figure 7.5 show, the rank of faulty method “lang3.time.FastDateFormat-820” changes from 17 to 3.

Additional considerations: Basic blocks and data-dependency graphs

Our re-ranking approach is based on integrating dynamic call and explicit data-dependency graphs. However, it is possible to consider only one of the call or data-dependencies. In the case of considering both, call and data-dependency relations have the same priority when inserting (see Section 7.2.2). This setting may change the number of parents and grandparents. Depending on the bug type, considering data-dependency in addition to the call graph might improve or harm the results. An example of improvement is Lang-9 (see Figures 7.6 and 7.7) where data-dependency links the most suspicious and faulty methods. On the other hand, adding more connections may increase the number of parents that are considered in re-ranking, therefore negatively impact the results, as it happens in Chart-7 (see Figures 7.8 and 7.9).

Besides, there is another option which is considering basic blocks. Two methods that are called from within one basic block have necessarily the same coverage profile. Therefore, it is not possible to discriminate between them. We can assign them the same rank or assign the parent a higher rank than the child. Assigning ranks to basic blocks will improve the results. In our basic setting, we just inform users which methods are in the same basic block utilizing a thicker arrow between them. The thick arrow between “lang3.time.FastDateFormat-TwoDigitNumberField-1355” and “lang3.time.FastDateFormat-TwoDigitNumberField-1348” means that they are in the same basic block.


```

src/main/java/org/apache/commons/lang3/time/FastDateParser.java [CHANGED]
@@ -141,6 +141,9 @@ private void init() {
141 141     currentFormatField= nextFormatField;
142 142     currentStrategy= nextStrategy;
143 143 }
144 +   if (patternMatcher.regionStart() != patternMatcher.regionEnd()) {
145 +       throw new IllegalArgumentException("Failed to parse \""+pattern+"\" ; gave up at index "+patternMatcher.regionStart());
146 +   }
144 147   if(currentStrategy.addRegex(this, regex)) {
145 148       collector.add(currentStrategy);
146 149   }

```

Figure 7.6.: Human Patch to Fix Lang-9

7.2.2. Parameter setting

Our idea on re-ranking is based on our observation that SBFL techniques might not always be accurate in localizing the fault in Top-10 ranks, but the most suspicious method s^* may be a hint that the fault is nearby. Parnin and Orso [136] performed an experiment to investigate how automated debugging can help programmers. They reported that programmers might use the FL tool just to identify the starting point for their investigations, some of which may be near the fault. Based on their data, programmers do not visit each element in the ranking list in a linear fashion. In addition, they reported that giving the ranking list to the programmers is not enough. They need more context, such as the structure information of the program.

Thus, one goal in our technique is helping the developers by pointing to the direction they need to traverse between methods. To find the right direction, we should look for the most prevalent patterns of the locality of s^* to the method that contains the fault e^* . Figure 7.10 shows some of these patterns. To translate these patterns into search strategies, we defined two parameters: Range and Mode. If we consider a call graph (or a data-dependency graph) where nodes are methods and edges are method calls, then:

Range shows the number of nodes between s^* and e^* . Three values for Range are shown in Figure 7.11. Regardless of direction, Range 1 includes all the direct neighbors of s^* , meaning its parents and its children. Range 2 includes direct neighbors and the neighbors of direct neighbors and so on. **Mode** shows the path direction to reach e^* starting from s^* . We defined four values for Mode. Figure 7.12 shows four Modes in Range 2.

- **Ancestor:** Going upward. Consider only parents, grandparents, etc.
- **Descendant:** Going downward. Consider only children, grandchildren, etc.
- **Chain:** Going both upward and downward. Include Ancestors and Descendants.
- **All:** Consider all directions in the defined Range. Include Chain and Siblings.

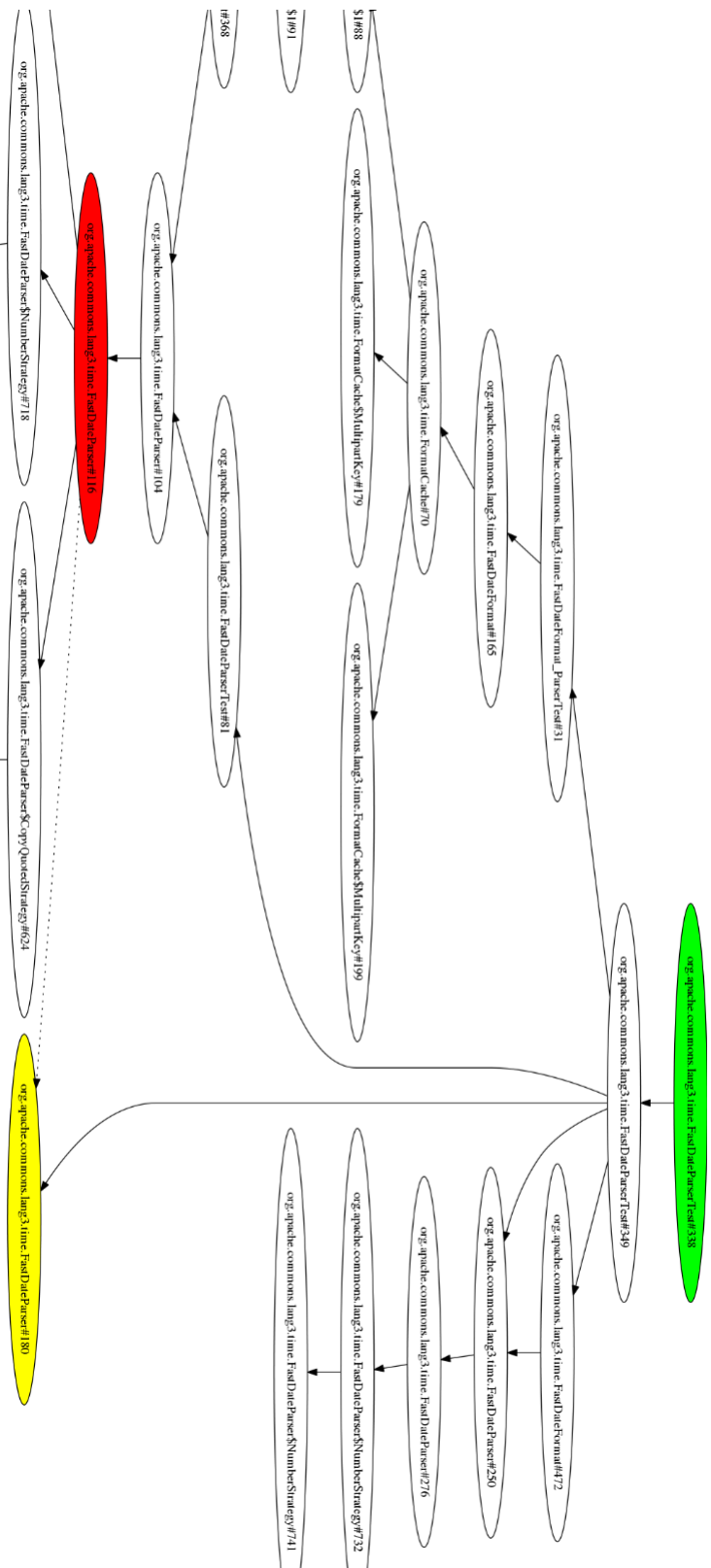


Figure 7.7.: Lang-9 Combined Call and Data-dependency Graph. A Data-dependency Edge Links the Faulty Node FastDate#116 to the Suspicious Node FastDate#180

```

source/org/jfree/data/time/TimePeriodValues.java | CHANGED
@@ -297,9 +297,9 @@ private void updateBounds(TimePeriod period, int index) {
297 297     }
298 298
299 299     if (this.maxMiddleIndex >= 0) {
300 -         long s = getDataItem(this.minMiddleIndex).getPeriod().getStart()
300 +         long s = getDataItem(this.maxMiddleIndex).getPeriod().getStart()
301 301         .getTime();
302 -         long e = getDataItem(this.minMiddleIndex).getPeriod().getEnd()
302 +         long e = getDataItem(this.maxMiddleIndex).getPeriod().getEnd()
303 303         .getTime();
304 304         long maxMiddle = s + (e - s) / 2;
305 305         if (middle > maxMiddle) {

```

Figure 7.8.: Human Patch to Fix Chart-7

7.3. Experiment

We can now refine our research questions.

7.3.1. Research Questions

We need to answer the following questions to evaluate how successful our solution is in improving SBFL:

- **RQ1:** Which combination of Range and Mode parameters is the most effective in this setting? Since we considered multiple approaches for the search around the most suspicious method, it is necessary to determine which combination of parameters works the best. The best setting points in the best direction to start looking for the faulty element.
- **RQ2:** How much improvement is achievable using the re-ranking? How does the re-ranking approach compare to the traditional SBFL and causal inference-based techniques?

7.4. Results

In this section, we describe the evaluation results, practical implications, and the threats to validity.

7.4.1. Evaluation Results

RQ1: Considering 4 values for Mode and 6 values for Range, we ended up constructing 24 re-ranking approaches. Our previous experiences in Chapters 6 and 4 and other studies

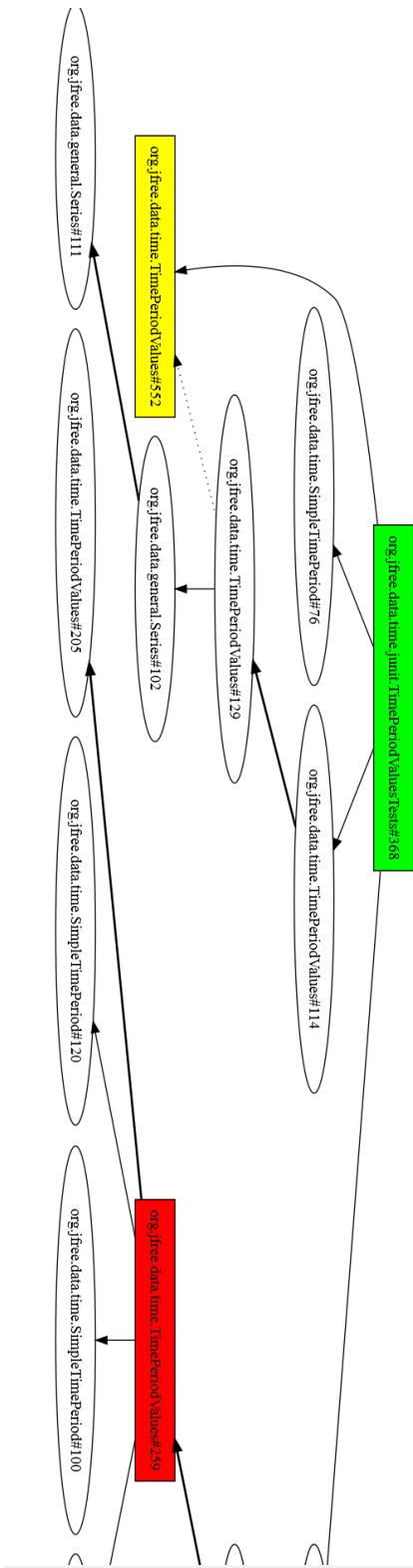


Figure 7.9.: Chart-7 Combined Call and Data-dependency Graph.

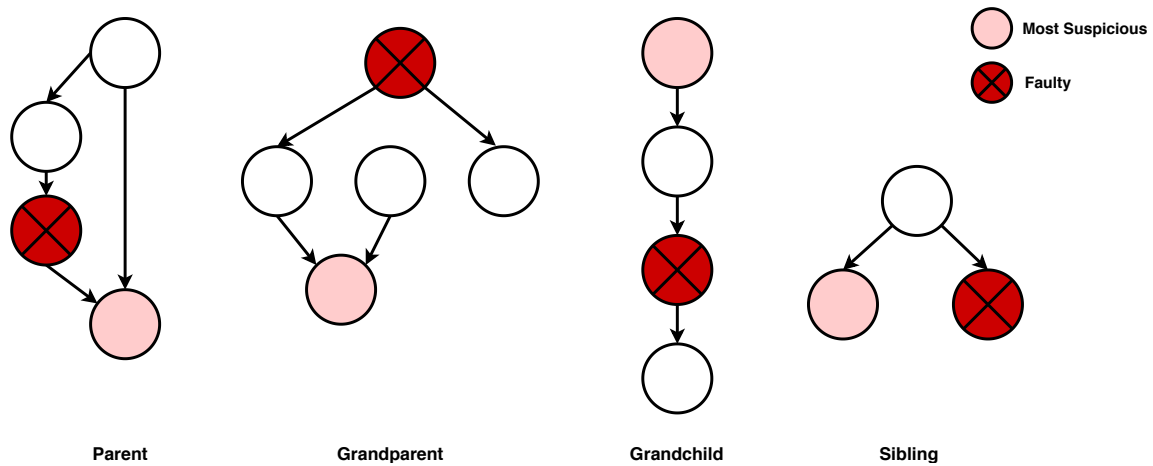


Figure 7.10.: Prevalent Patterns of Locality of Suspicious Method s^* to the Method that Contains a Fault e^*

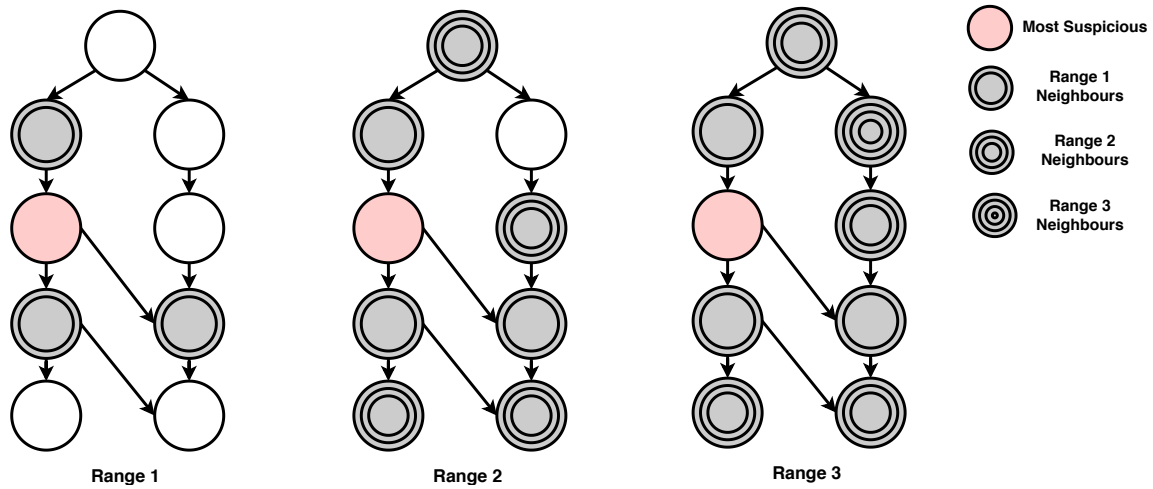


Figure 7.11.: Different Ranges; Steps Between the Most Suspicious and the Faulty Elements

such as [185] showed that DStar results are slightly better than other traditional metrics. Thus, we apply re-ranking on DStar2 results.

We considered 183 Defects4J bugs from projects Chart, Closure, and Time as our training data. The results from running top-6 combinations on the training data can be found in Table 7.1. As the table shows, the differences between top-6 combinations are not huge. Across all combinations, Ancestor-Range2, followed by Chains-Range2, are the two best performing approaches. It means that for most of the buggy versions in Defects4J, if Dstar2 s^* does not contain a fault, usually one of its parents or grandparents contains a fault. In other words, a developer checking the code to find the fault should first inspect the most

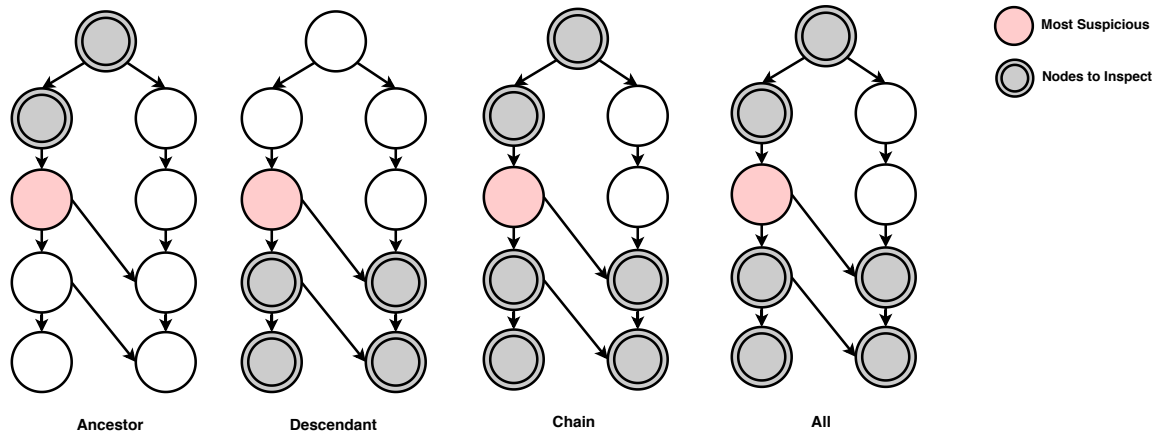


Figure 7.12.: Different Modes; Direction of the Path Between the Most Suspicious and the Faulty Elements

suspicious method then should search upward on the call graph to find and inspect its ancestors. This is more effective than linearly inspecting the ranking list. We consider the Ancestor-Range2 combination to be our best result since it shows the best overall results. The top-performing value for each evaluation metric is highlighted in gray.

RQ2: As was the case in RQ1, Ancestor-Range2 is the best performing combination for our re-ranking approach. When compared to the raw DStar2 results, it exhibits better EXAM and Top-10 scores as shown in Table 7.2. The results show an improvement in Top-10 scores by 8 and 3 percentage points in Chart and Closure respectively. Overall, all scores have improved, however slightly.

In comparing all existing metrics and the best combination from RQ1, Ancestor-Range2 exhibits better EXAM and Top-10 scores. It consistently outperformed all other combinations across Median and Top-10 metrics. Only in Chart and Time, Ochiai has slightly better Mean scores. Due to this, we consider Ancestor-Range2 as a more effective technique when compared to the popular SBFL techniques and also casual inference-based techniques.

As seen in RQ1, our Ancestor-Range2 re-ranking approach is more effective than other techniques. To measure its effectiveness on unseen test data, we apply it on 63 bugs from Lang project. The results are shown in Table 7.3. Re-ranking strategy Chain-Rang3 shows better results than Ancestor-Range2. In general, re-ranking strategies show better results than SBFL and causality-based techniques, however slightly.

Our experiment results show slight improvements. There are several works in the literature suggesting improving FL via either a new technique such as [196] and [141] or optimizing the test suite such as [143] and [187]. Based on the results of our experiment and these other studies, it seems that improving the effectiveness of FL is limited and the improvement ratio varies from one project to another. Thus, in order to further improve it, we need to understand the reasons why it is sometimes ineffective. To this goal, we conduct

another experiment in Chapter 8.

Table 7.1.: RQ1 - Re-ranking Results Using Different Modes and Ranges. μ : mean EXAM, M: median EXAM, Top-10 (%)

	Chart			Time			Closure		
	μ	M	Top-10	μ	M	Top-10	μ	M	Top-10
Chains-Range1	10.2	2.00	84	19.19	6.00	63	115.54	21.00	37
Chains-Range2	10.10	2.00	88	19.98	5.00	56	118.92	24.00	34
Chains-Range3	10.40	2.00	84	23.96	5.00	56	127.20	30.00	29
Ancestors-Range2	13.84	2.00	84	18.69	4.00	70	115.86	19.00	40
Descendants-Range2	11.62	2.00	76	19.22	5.00	67	119.29	24.50	30
All-Range2	10.22	2.00	84	19.44	5.00	67	117.50	21.00	34

Table 7.2.: RQ1 - Fault Localization Effectiveness - μ : mean EXAM, M: median EXAM, Top-10 (%)

	Chart			Time			Closure		
	μ	M	Top-10	μ	M	Top-10	μ	M	Top-10
Ancestors-Range2	13.84	2.00	84	18.69	4.00	70	115.86	19.00	40
DStar	15.12	2.00	76	18.06	3.50	70	117.03	19.50	37
Ochiai	8.62	2.00	84	17.06	4.50	67	119.97	21.00	37
Tarantula	11.86	2.00	84	20.52	6.00	67	126.37	21.00	35
MFL	11.86	3.00	80	25.20	7.00	70	463.20	500.00	8
Improved-MFL	24.06	3.00	72	23.11	4.00	59	246.08	40.00	27

Table 7.3.: RQ3 - Evaluation of Re-ranking Approach on Test Data Project Lang - μ : mean EXAM, M: median EXAM, Top-10 (%)

	μ	M	Top-10
Chain-Range1	4.29	2.00	89
Chain-Range2	3.67	2.00	92
Chain-Range3	3.54	2.00	94
Decendant-Rang2	4.42	2.00	87
All-Range2	3.64	2.00	92
Ancestor-Range2	3.88	2.00	90
DStar	4.62	2.00	87
Ochiai	4.65	2.00	89
Tarantula	5.45	2.00	89
MFL	4.48	2.00	90
Improved-MFL	4.14	2.00	89

7.4.2. Threats to Validity

We evaluated our approach on the 246 bugs in the Defects4J data set. While we do not claim that our results can be generalized to bugs in the field, Defects4J is the largest available

```
@Test
public void test(){
    boolean result = unit.shouldReturnTrue();
    if (!result) {
        Assert.fail(unit.getName() +
            "returned wrong value");
    }
}
```

Figure 7.13.: Triggering a Method After Test Failure

database of real bugs for Java. However, it is possible that our findings may be restricted to Defects4J.

As mentioned earlier, the quality of the test suite impacts the results of any FL technique. The quality of Defects4J test suites may not reflect the quality of other projects in the real world. Thus, the results using our techniques may not be similar to what we have produced.

While a majority of the research into FL focuses on programs with a single bug, this is not always the case for real world software. Nevertheless, this is also the case for Defects4J. Each buggy version in Defects4J contains only one bug.

When evaluating our approach, we made the simplifying assumption of perfect bug detection, in that, if given a program element containing a fault, the user will be able to localize the fault 100% of the time [185]. This allows us to evaluate the effectiveness of each FL technique, without having to worry about the complexity of the faults themselves. However, in practice, understanding a fault to repair it is also difficult and time-consuming.

7.5. Practical Implications

Overall, re-ranking can improve SBFL results especially for complex projects like Closure. However, the improvement might not be enough to make it applicable in practice right away. As the first step in our journey toward exploring the reasons behind FL ineffectiveness, we looked deeper into some of the case that our approach could not improve the results. Based on our observation, we propose the following technical take-away that. We do not claim that these technical issues are the only reasons behind FL ineffectiveness. However, they have a negative impact on the result of any FL technique.

7.5.1. Triggering a method after test failure

We observed several times that a method is being triggered immediately after tests are failed. Consequently, FL results are skewed toward the triggered method, disturbed from the fault itself. An example is shown in Figure 7.13.

As shown in Figure 7.13, method `unit.getName()` is triggered, if the test result is not as expected. An example of this pattern is Lang-10, where all FL techniques performed poorly

(see Figure 7.14). Executing more code after a test already asserted as a failure skews the FL results toward giving a larger subset of the code the same suspiciousness rank. Therefore, failures should be reported as soon as they are detected.

```
337     @Test
338     public void testLANG_831() throws Exception {
339         testSdfAndFdp("M E", "3 Tue", true);
340     }
341
342     private void testSdfAndFdp(String format, String date, boolean shouldFail)
343         throws Exception {
344         Date dfdp = null;
345         Date dsdf = null;
346         Throwable f = null;
347         Throwable s = null;
```

Figure 7.14.: Lang-10. Calling testSdfAndFdp() After the Test Failure Makes It Wrongly the Most Suspicious Method

Combining multiple tests in one test method

One common problem in Defects4J projects is that multiple test cases are combined in one test method. Two examples are shown in Figures 7.15 and 7.16. In the first example, the developer has not created a test method for every simple test case. While this makes the code more readable, it has a bad impact on FL results. The whole method is counted as a single test case. Therefore, the coverage and call graph are computed for the whole method. By splitting these test cases, the coverage profile is more accurate, and it is easier to detect the fault. The solution JUnit offers to test the same method with different parameters is called “parametrized tests”². The developer should write one method generating the input, and a test method that is called with every specified input.

The second example also has a negative impact on FL results. As shown in Figure 7.16, the first test case is to inspect if *unit.setValue(1)* works as expected. Then the same unit is re-used to test another method. In this case, the coverage profile and the hit spectrum contain one test case. However, there are two unit tests. An example of these patterns is Lang-9. Xuan and Monperrus [187] proposed an algorithm to split the test cases at every assertion automatically.

7.5.2. Combining unit and integration test suites

As mentioned in RQ4, existing FL techniques perform better for unit test. However, manual FL is more challenging in integration testing. Therefore, automated FL techniques are in higher demand in integration testing. In order to improve FL results for integration

²<https://github.com/junit-team/junit4/wiki/parameterized-tests>

```
@Test
public void test(){
    assertEquals("A", unit.upperCase("a"));
    assertEquals("B", unit.upperCase("b"));
    assertEquals("C", unit.upperCase("c"));
}
```

Figure 7.15.: Combined Unit Tests

```
@Test
public void test2(){
    int a = unit.setValue(1);
    assertEquals(0, a);
    int b = unit.calculate(a);
    assertEquals(1, b);
}
```

Figure 7.16.: Combined Unit Tests

testing, we propose combining unit and integration test suites. Adding unit tests to the set of integration tests can decrease the indistinguishability of the test suite. Integration tests cover a large subset of methods inevitably. Adding unit tests increase the variety in the execution profiles.

7.6. Conclusion

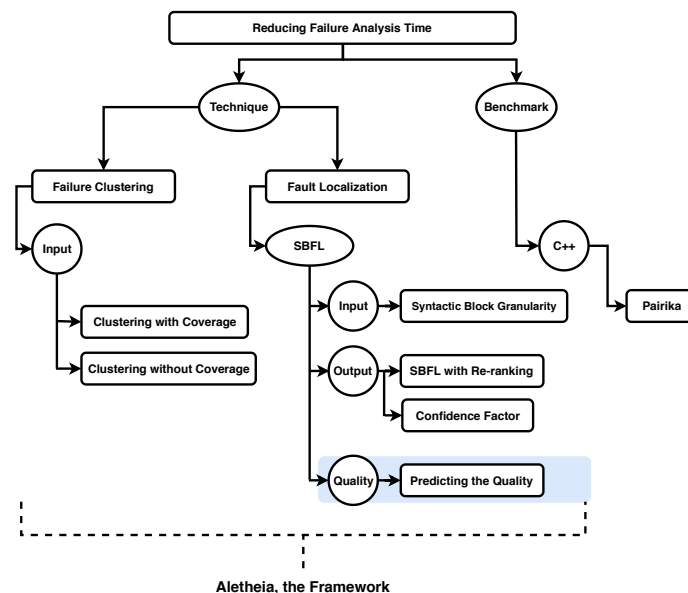
Despite ongoing research efforts, state-of-the-art SBFL techniques are not applicable in practice yet. In this chapter, we proposed a re-ranking strategy which leverages dynamic call and data-dependency graphs of failing executions to improve SBFL techniques. Our experiments showed that to localize a bug, a developer should first look for the most suspicious method on the call graph then look upward and inspect its parent and grandparents instead of inspecting the ranking list in a linear fashion. Our re-ranking strategy outperformed the traditional SBFL techniques and also causal inference-based techniques.

While we cannot report on a second study in a different domain here and clearly see the threats to external validity, we ourselves are sufficiently confident to repeat our own study in a different context. Moreover, we believe that our achieved relative improvement is not enough for the real-world, but it adds to the growing body of evidence about the potential usefulness of fault localization techniques.

Based on our experiment and observations, we believe that the next steps in improving the effectiveness of FL should be improving the quality of the code via finding the metrics that influence the effectiveness of SBFL. As our first practical step toward this goal, we suggested some technical changes that can improve the effectiveness of SBFL.

8. Predicting the Quality of SBFL

This chapter presents a model to predict the quality of spectrum-based fault localization. Considering the prediction, developers can decide whether to use the fault localization tool or not.



We saw in previous chapters that despite ongoing research in the area of SBFL, there are still many unanswered questions. Two of the most important questions that have not been answered are: Why are these techniques sometimes effective and sometimes ineffective? What are the metrics that influence the effectiveness of FL? In this chapter, we try to answer these questions by collecting 70 Static, Dynamic, Test suite, and Bug related metrics. Our analysis results show that a combination of 4 Static, 4 Dynamic, and 2 Test metrics, gives us a model with excellent discrimination power (AUC=0.86) which can be used in 2 ways: as a prediction model for the effectiveness of FL, and as a confidence factor for the results of FL. These 10 metrics are the most influential metrics on the effectiveness of FL. Thus, they can be used as a guide to improve the code quality for more effective FL.

8.1. Introduction

In previous chapter, we saw that what is observable from our experiments and other studies on Defects4J is twofold. First, FL is not equally effective on all of Defects4J's projects. Second, the improvement ratio achieved by various techniques varies from one project to another (see Chapters 6 and 7 and [166, 143] for instance).

In this chapter, we raise the question of why on the two observations mentioned above. We try to understand why sometimes FL is effective and sometimes not. Is the reason in the characteristics of the code, the test suite or the type of bugs? We are trying to learn the metrics that influence the effectiveness of FL. Understanding these characteristics can help in two ways. First, we would know how to improve the code to facilitate FL. Second, we would be able to predict the effectiveness of FL. This helps us in deciding where and when to use it to avoid wasting time and money.

Among the Defects4J projects, Closure is the project with the largest integration tests and also the one with the least FL effectiveness. Nevertheless, for some buggy versions of this project, FL is quite effective and successful. Thus, the effectiveness is not ruined only by the high coverage of the test or complexity of the code. Then the question would be what aspects of the projects are the most influential on the effectiveness of FL? The static characteristic of the source code? Dynamic measures of the code? Test suite characteristic? Or the bug location and bug type?

To answer these questions, we collected four groups of metrics: Static metrics, Dynamic metrics, Test-suite metrics, and Bug characteristics. Static metrics measure the static aspects of the whole source code. Dynamic metrics measure the dynamic aspects considering the test runs. Test suite metrics are correlated to the test suite. Bug characteristics define the repair patch of the bug and also the location of the bug on the call graph of failing tests. In total, we collected 70 metrics and computed them for the buggy versions of five Defects4J projects.

We utilized SBFL to compare the effectiveness of FL on different projects. If the faulty block gets a very high rank in the ranking list, we say that FL is effective. Using collected metrics as variables or features and SBFL effectiveness as labels, we formed a data set. We did regression analysis and classification on the data set with the goal to find the most relevant and influential metrics. The metrics that can separate an effective FL from an ineffective one, can be used to generate a model for prediction.

The results show that among all the collected metrics, 4 static metrics (% Methods with LoC>30, % Methods with Nesting Depth>5, % Methods with $3 \leq \text{Nesting Depth} \leq 5$, Mean # of Fields per Type), 4 Dynamic metrics (Mean Node Degree, Max. Node Out-Degree, Graph Diameter, Response for Class), and 2 Test metrics (% Method Coverage, % Methods Covered in Failing Tests) are the most influential metrics. Our analysis results also show that the bug location is more influential than the bug type. The Logistic regression model generated on the influential metrics to predict the effectiveness of FL, has an excellent discrimination power between effective and ineffective FL (AUC=0.86). Thus, this model can be used a priori as a prediction model to predict the effectiveness of FL, and a posteriori

to predict the confidence of FL results.

The prediction model can be used in the following ways. In the first scenario, the developers check the prediction results, if the effectiveness probability is very low, they avoid running the FL tool. If the effectiveness probability is not very low, they refactor their code to facilitate fault localization. Then, again they check the effectiveness probability. If the effectiveness probability is high, they run the FL tool to get the hints about the location of the bug. In the second scenario, developers run the FL tool in the debugging session. They get the results aligned with a confidence factor. If the confidence score is high, they trust the results and continue using the localization results. Otherwise, they avoid using the results and inspect the code manually.

As a recap, the **problem** is that in spite of a lot of work put in these techniques, existing techniques are not effective in practice yet. Introducing new techniques and ideas have not solved the problem yet. What are the influential aspects of a project on predicting the effectiveness of FL? How can we improve our project to facilitate FL? Is it possible to predict the effectiveness of FL?

As a **Solution**, we try to learn the influential metrics on the effectiveness of automated FL. We believe that by learning these factors, we will be able to improve our projects and facilitate FL. Besides, using these metrics, we will be able to predict the effectiveness of FL and avoid using it if it is going to be ineffective.

Thus, in this chapter:

- We explore the correlation between a collection of static, dynamic, test, and bug metrics and the effectiveness of FL.
- Using the most influential metrics, we introduce a model that can be used for predicting the effectiveness of FL and for assigning a confidence factor to the results of FL.

8.2. Methodology

In Sections 8.3 and 8.4, we describe our methodology. First, we explain the steps in generating and populating the study data set. Then, we describe our data analysis steps.

Utilizing our generated data set, our goal is answering the following **Research Questions**:

- **RQ1:** Which group of Static, Dynamic, or Test metrics is the most influential in predicting the effectiveness of FL?
- **RQ2:** Which metrics are the most influential in predicting the effectiveness of FL? What is their impact on the effectiveness of FL?
- **RQ3:** Can we generate a model on the most influential metrics to predict the effectiveness of FL? Can we generate a model using the most influential metrics to assign a confidence factor to the results of FL?

8.3. Generating the Data Set

To identify the factors that influence the effectiveness of FL, we utilized regression analysis and machine learning predictive modeling, namely classification. Our goal is identifying variables in our data that are most relevant to the predictive modeling we are working on. In the following, we explain the steps in generating and populating our labeled data set for regression analysis and classification.

8.3.1. Variables

The variables of the data set are the metrics for which we want to measure the influence on the effectiveness of FL. We identified and collected four groups of metrics described in the following. The criteria for metric selection was twofold. First, we can have a hypothesis about why the metric influences the effectiveness of FL. Second, the metric is actionable meaning a developer can improve the values with a good programming style. Therefore, we did not include metrics such as “Lines of Code” that is inevitably large in large projects.

Static Metrics. Static metrics measure static characteristics of the source code. These metrics do not include any run time or any bug related values. Table 8.1 shows the static metrics and their short descriptions. We used Teamscale¹ and Codepro² tools to compute the Static metrics.

Table 8.1.: Static Metrics

ID	Metric	Definition
S1	PML	% of Methods with LoC>30
S2	PHFS	% of Files with LoC>750
S3	PMFS	% of Files with 300<LoC<750
S4	PHND	% of Methods with Nesting Depth>5
S5	PMND	% of Methods with 3<=Nesting Depth<=5
S6	CC	Mean Cyclomatic Complexity [121]
S7	MCC	# of Methods with 10<CC<20
S8	AFFC	Mean Afferent Coupling [18]
S9	EFFC	Mean Efferent Coupling [18]
S10	ABKD	Mean Block Depth
S11	ADIH	Mean Depth of Inheritance Hierarchy
S12	ALOCPM	Mean # of LoC per Method
S13	ANOCPT	Mean # of Constructors per Type
S14	ANOFPT	Mean # of Fields per Type
S15	ANOMPT	Mean # of Methods per Type

Dynamic Metrics. Dynamic metrics are the metrics measured at run-time. In addition, we generated the call graph of tests and collected some of the graph metrics as Dynamic metrics. The list of these metrics is shown in Table 8.2. However, we are aware that

¹ Available here: <https://www.cqse.eu/en/products/teamscale/landing/>

² Plugin available here: <https://dl.google.com/eclipse/inst/codepro/latest/3.7>

the impact of the test suite on dynamic values is inevitable. We developed a tool called `jdcallgraph` to generate dynamic call graphs ³.

Table 8.2.: Dynamic Metrics

ID	Metric	Definition
D1	VC	# of Nodes in Call Graph ⁴
D2	EC	# of Edges in Call Graph ⁵
D3	MAXVD	Max. Node Degree ⁶
D4	MVD	Mean Node Degree
D5	MAXVI	Max. Node In-Degree ⁷
D6	MVI	Mean Node In-Degree
D7	MAXVO	Max Node Out-Degree ⁸
D8	MVO	Mean Node Out-Degree
D9	MSND	Avg. Start Node Degree
D10	GD	Graph Diameter ⁹
D11	GR	Graph Radius ¹⁰
M12	MGD	Mean Geodesic Distance ¹¹
D13	VCON	Node Connectivity ¹²
D14	ECON	Edge Connectivity ¹³
D15	CICo	Mean Clustering Coefficient ¹⁴ [112]
D16	SCOV	% of Statement Coverage
D17	CBO	Coupling Between Objects [18]
D18	RFC	Response for Class [18]

Test Suite Metrics. Test suite metrics measure some characteristics of the test suite. For each faulty program version, we generated a hit spectrum [76], a matrix which shows which methods were executed in different test runs (see 8.3.2). Test suite metrics are measures based on the generated hit spectra. Table 8.3 demonstrates these metrics.

Table 8.3.: Test suite Metrics

ID	Metric	Definition
T1	T	# of Tests
T2	M	# of Methods
T3	PPT	% of Passing Tests
T4	PFT	% of Failing Tests
T5	D	Density [143]
T6	G	Diversity [143]
T7	U	Uniqueness [143]
T8	DDU	Density×Diversity×Uniqueness [143]
T9	MatSpar	Matrix Sparsity
T10	MetCov	% of Method Coverage
T11	COVPT	% of Methods Covered in Passing Tests
T12	COVFT	% of Methods Covered in Failing Tests
T13	AVGMV	Mean Covered Methods per Test

Bug Metrics. Utilizing bug characteristics, we try to characterize the bugs. To this goal,

³Available here: <https://github.com/dkarv/jdcallgraph>

we considered the patches that have been used to repair the bug and defined some metrics based on them. In addition, we considered the faulty nodes on the generated call graphs of failing tests and computed some metrics based on the location of the faulty node on the call graph. Although these metrics are not helpful in predicting the effectiveness of the FL (since bugs are not known a priori), we include them in our experiment to measure the impact of *fault type* and *fault location* on the effectiveness of FL. Table 8.4 lists the bug metrics. We extracted metrics B1 to B8 and B14 to B24 from [163].

Table 8.4.: Bug Metrics

ID	Metric	Definition
B1	NF	# of Files Changed in Patch
B2	NC	# of Classes Changed in Patch
B3	NM	# of Methods Changed in Patch
B4	NL	# of Lines Changed in Patch
B5	NCH	# of Chunks Changed in Patch
B6	NRP	# of Repair Patterns ¹⁵
B7	NRA	# of Repair Actions ¹⁶
B8	EXCP	Exception Type
B9	FaultyD	Mean Degree of Faulty Node
B10	FaultyInD	Mean In-Degree of Faulty Node
B11	FaultyOutD	Mean Out-Degree of Faulty Node
B12	FaultyCTRL	Degree Centrality of Faulty Node ¹⁷
B13	SpreadCode	# of Lines Between Patched Lines
B14	Add	# of Added Lines
B15	Mod	# of Modified Lines
B16	Rem	# of Removed Lines
B17	MSIGCH	# of Changed Method Signature
B18	MCARGCH	# of Changed Method Call Args.
B19	AMCAL	# of Added Method Calls
B20	CHCOND	# of Changed Conditions
B21	AV	# of Added Variables
B22	AIF	# of Added If-Statements
B23	AL	# of Added Loops
B24	AM	# of Added Methods

8.3.2. Target Class

As mentioned earlier, we defined our problem as a regression problem or supervised machine learning problem. To define the target labels or class labels, we used method-level SBFL results.

Kochhar et al. [94] report that practitioners find SBFL results useful if it ranks the faulty method in Top-5 or Top-10 ranks. Therefore, initially, we considered two settings for our labeling. In the first setting, if the rank of the faulty method is between 1 to 10, we label it as “effective” or “1”, otherwise it gets “ineffective” or “0” label. In the second setting, if the faulty method has a rank between 1 to 5, it gets “effective” label, if it is between 5 to 10, it

gets “acceptable” rank. Otherwise, it gets “ineffective” label. However, early results using the second setting were so discouraging that we did not pursue this avenue any further.

It is worth to mention that, we tried other well-known similarity formulæ such as Tarantula [76] and DStar [181] as well. As our experiment and other experiments [141] show, there is not any significant difference between these formulæ. Especially since we are considering ranges of ranks for assigning labels, our strategy can tolerate small differences. In fact, our experiment is not sensitive to the similarity formula used.

8.3.3. Populating the Data Set

As mentioned earlier, we selected Defects4J as our study benchmark. Each instance of data in our data set is a faulty version extracted from Defects4J. We computed our metrics on Defects4J projects to generate data values and performed SBFL on each version to evaluate the effectiveness of SBFL on it and generate the class labels. Figure 8.1 demonstrates a summary view of our data set, its records, variables, and class labels. Our final data set has 341 instances and 70 variables. The “effective” label has been assigned to 193 instances¹⁸.

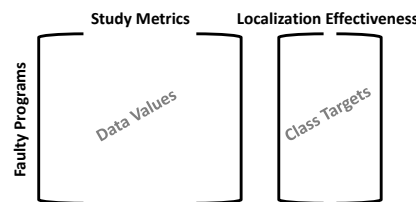


Figure 8.1.: Study Data Set

8.4. Data Analysis

To measure the correlation between our metrics and the effectiveness of FL, we use regression model and five other classification techniques. In the following, we explain our analysis steps.

8.4.1. Pearson’s Correlation

One assumption in regression models is that there is no high inter-correlations among the predictors. Tabachnick and Fidell [169] suggest that as long as the correlation coefficients among independent variables are less than 0.90 the assumption is met. Thus, first, we need to find and remove correlated metrics. To this end, we use Pearson correlation [140] which

¹⁸Our data set is available here: <https://figshare.com/s/c41ae04bbf0976cded8b>

is a number between -1 and 1 that indicates to which extent two metrics are linearly related. A Pearson correlation between metrics X and Y is calculated by:

$$r_{XY} = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}}, \quad (8.1)$$

where n is the number of instances, and $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$; analogously for \bar{Y} .

Table 8.5 shows the list of metrics that the correlation coefficient among them is larger than 0.9. Thus, for further analysis, we must keep only one from each group.

Table 8.5.: Correlated Metrics - (Correlation > 0.90)

Group	Correlated metrics
Static	(S5,S7,S9)
Dynamic	(D1,D2,D3,D5,D7,D12)(D4,D6,D8)
Test	(T5,T9)(T7,T8)(T2,T13)(T10,T11)(T3,T4)
Bug	(B1,B2)(B11,B12)

8.4.2. Logistic Regression

We use regression analysis to describe the relationship between the FL effectiveness and the collected metrics. Since the outcome variable, effective or not, is dichotomous, we utilize Logistic regression [26]. The goal of an analysis using this model is to find the best fitting and most interpretable model to describe the relationship between outcome variable (effective or ineffective in our case) and a set of independent variables (metrics) [29].

Logistic function, on which the logistic model is based, provides estimates that are in the range between 0 and 1. It also provides an S-shaped description of the combined effect of the metrics on the effectiveness of FL [90].

The probability of being effective ($D=1$) can be denoted by the conditional probability statement $P(D = 1|X + 1, X_2, \dots, X_k)$. For convenience, we denote it as $P(\mathbf{X})$ where the \mathbf{X} is a notation for the collection of metrics X_1 through X_k . Given the X s, the model is defined as:

$$P(\mathbf{X}) = \frac{1}{1 + e^{-(\alpha + \sum \beta_i X_i)}} \quad (8.2)$$

The terms α and β_i in this model represent unknown parameters that we need to estimate based on data obtained on the X s and D for a group of instances. The general method of estimation is called *maximum likelihood* [90]. The algorithm of maximum likelihood determines the regression coefficient for the model that accurately predicts the probability of the binary dependent variable.

Evaluation Metric

To evaluate a Regression model, we need to evaluate the significance of coefficients, the significance of the model, and the goodness of fit [90].

Significance of Coefficients. After estimating the coefficients, first, we need to assess the significance of the variables in the model. This usually involves testing a statistical hypothesis to determine whether the independent variables in the model are “significantly” related to the outcome variable. To this goal, we use the Likelihood Ratio Test (LRT) [90] along with the associated p-Value for the Chi-squared distribution. The LRT is computed as:

$$LRT = -2 \times (\text{LogLikelihood1} - \text{LogLikelihood0}), \quad (8.3)$$

where LogLikelihood1 is the log-likelihood for the model with one independent variable and LogLikelihood0 is the log-likelihood for the null hypothesis model with only the constant α . Under the hypothesis that β_1 is equal to zero, LRT follows a Chi-squared distribution with 1 degree of freedom. This means, for each independent variable, first we fit a model containing only the constant term. Next, we fit a model containing the independent variable along with the constant. Then, we measure the LRT. If the p-Value ≤ 0.05 , we conclude that the variable is significant.

Table 8.6 gives the **LRT** results along with the associated **p-Values**. Column +/- shows the coefficient sign which demonstrates the negative or positive impact of the coefficient on the probability of being “effective”. As the results show, 25 metrics are not significant. Among the correlated metric groups, we pick the most significant ones as the representatives of the groups. Thus, we pick S5, D4, D7, T9, T7, T13, T10, B2, and B12 from correlated metrics for further analysis.

Significance of model. The LRT for the overall significance of the coefficients in the model is performed in the same way as the univariable case performed in the previous step. First, we fit a model containing only the constant term. Next, we fit a model containing all the significant independent variables along with the constant. Then, we measure the LRT. If the p-Value > 0.05 , we conclude that at least one or more coefficients are different from zero. The degree of freedom for the Chi-square test is equal to the number of metrics considered in the model. The log-likelihood for the constant only model is equal to -233.3853.

Static. The fitted Regression model on Static metrics is shown in Table 8.7. **Coeff.** is the estimated coefficient value. The larger the coefficient in absolute value, the stronger the impact of the metric on the probability of being “effective” or “1”. **Std. Err.** is the standard error of the estimated coefficient. **tStat** is t statistic for a test that the coefficient is zero. And **p-Value** is the p-Value for the t statistic.

The LRT score for this model is equal to $-2 \times (-233.3853 - (-191.0476)) = 84.6754$ and $P[\chi^2(8) > 84.6754] < 0.00001$ is the p-Value for the test, which is significant. Thus, at least one or more of the coefficients are different from zero. Before concluding that any or all of the coefficients are nonzero, we should look at the univariable t test statistics shown in the fourth column. Under the hypothesis that an individual coefficient is zero, these statistics

Table 8.6.: Significance of the Coefficients - $P = P[X^2(1) > LRT]$

Metric	+/-	LRT	P	Metric	+/-	LRT	P
S1	-	24.717	<.00001	T3	-	1.487	.2226
S2	-	3.490	.0617	T4	+	1.487	.2226
S3	+	0.126	.7221	T5	-	2.9399	.0864
S4	-	36.4030	<.00001	T6	-	19.585	<.00001
S5	-	58.2877	<.00001	T7	+	19.141	<.00001
S6	-	64.228	<.00001	T8	+	9.762	.0017
S7	-	38.085	<.00001	T9	-	4.420	.0298
S8	-	1.6732	.1958	T10	+	7.378	.0066
S9	-	56.907	<.00001	T11	+	6.810	.0090
S10	+	22.075	<.00001	T12	-	20.211	<.00001
S11	+	8.592	.0033	T13	-	78.959	<.00001
S12	+	1.8136	.1780	B1	-	2.767	.0962
S13	+	39.483	<.00001	B2	-	5.885	.0152
S14	-	50.973	<.00001	B3	+	0.018	.8929
S15	+	0.501	.4790	B4	+	0.574	.4484
D1	-	80.220	<.00001	B5	+	0.225	.634
D2	-	78.866	<.00001	B6	+	2.515	.1127
D3	-	89.130	<.00001	B7	-	0.028	.8650
D4	-	45.462	<.00001	B8	-	0.012	.9102
D5	-	71.043	<.00001	B9	-	43.312	<.00001
D6	-	45.487	<.00001	B10	-	56.820	<.00001
D7	-	100.715	<.00001	B11	-	84.230	<.00001
D8	-	45.487	<.00001	B12	-	91.122	<.00001
D9	-	70.290	<.00001	B13	+	0.127	.7215
D10	-	109.442	<.00001	B14	+	4.324	.0375
D11	-	0.0936	.7596	B15	-	0.052	.8196
D12	-	79.800	<.00001	B16	-	6.502	.0107
D13	+	37.623	<.00001	B17	+	0.015	.9015
D14	+	37.623	<.00001	B18	-	3.003	.0832
D15	+	0.006	.9342	B19	-	23.890	<.00001
D16	+	0.007	.9323	B20	+	3.315	.068
D17	-	74.731	<.00001	B21	-	5.8902	.0159
D18	-	65.496	<.00001	B22	-	9.096	.0025
T1	-	11.041	.0008	B23	-	0.342	.5582
T2	-	74.789	<.00001	B24	+	0.526	.4682

follow the standard normal distribution. The p-Values computed under this hypothesis are shown in the fifth column. If we use a level of significance of 0.05, then we can conclude that metric S4 is significant, metric S1 is somewhat significant, and the rest are not significant.

As our goal is to obtain the best fitting model while minimizing the number of parameters, the next step is to fit a reduced model containing only significant metrics and compare the reduced model to the full model containing all of the metrics. The results of fitting the reduced model are given in Table 8.8.

Table 8.7.: Fitted Logistic Regression Model of FL Effectiveness on Static Metrics - Full Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	4.1771	9.6357	0.4335	0.6646
S1	498.3868	272.6536	1.8279	0.0676
S4	30.5677	11.9219	2.5640	0.0103
S5	-2.09e-04	1.65e-04	-1.2634	0.2064
S6	-0.0135	0.0086	-1.5707	0.1162
S10	-4.4567	5.9831	5.9831	-0.7449
S11	1.3394	0.9508	0.9508	1.4087
S13	0.5271	1.1065	1.1065	0.4764
S14	-2.2365	1.2927	1.2927	-1.7301

Log-Likelihood=-191.0476

Table 8.8.: Fitted Logistic Regression Model of FL Effectiveness on Static Metrics - Reduced Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	2.6635	0.7836	3.3991	6.7597e-04
S1	-301.9209	145.8237	-2.0704	0.0384
S4	-22.9781	5.9107	-3.8875	1.0126e-04

Log-Likelihood=-212.9489

The difference between the two models is the exclusion of the metrics S5, S6, S10, S11, S13, and S14 from the first model. The LRT comparing these two models is obtained using the equation 8.3. The $LRT = 43.8026$ which, with 6 degrees of freedom (8-2), has a p-Value of $P[\chi^2(6) > 43.8026] < 0.00001$. As the p-Value is small, we conclude that the full model is better than the reduced model. It means, there is statistical justification for including S5, S6, S10, S11, S13, S14 in the model. However, S1 and S4 (% Methods with LoC>30 and % Methods with Nesting Depth>5) have the greatest impacts (Table 8.7).

Dynamic. The fitted Regression model on Dynamic metrics is shown in Table 8.9. Since $LRT = 135.2666$ with $P[\chi^2(7) > 135.2666] < 0.00001$, at least one or more of the coefficients are different from zero. P-Values of t test show that metrics D4, D7, D10, and D18 are significant and the others are not significant. Thus, we fit a reduced model using these metrics. The fitted model is shown in Table 8.10. The LRT of comparing reduced and full model is $LRT = 24.5796$ with $P[\chi^2(3) > 24.5796] < 0.00001$. Thus, again the full model is a better fitted model. All D4, D7, D9, D10, D15, D17, D18 are adding to the model. However, D4, D10, and D15 (Mean Node Degree, Graph Diameter, and Clustering Coefficient) have the greatest impacts (8.9).

Test. The fitted Regression model on Test metrics is shown in Table 8.11. The $LRT = 135.2720$ with $P[\chi^2(7) > 135.2720] < 0.00001$ shows that at least one or more coefficients are different from zero. The reduced model is shown in Table 8.12. The full model is slightly better than the reduced model ($LRT = 6.3828$ and $P[\chi^2(2) > 6.3828] = 0.041114$). Metrics

Table 8.9.: Fitted Logistic Regression Model of FL Effectiveness on Dynamic Metrics - Full Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	1.3551	0.4901	2.7647	0.0057
D4	0.9083	0.2991	3.0373	0.0024
D7	-0.0533	0.0238	-2.2362	0.0253
D9	0.0431	0.0270	1.5950	0.1107
D10	-0.2302	0.0568	-4.0510	5.1001e-05
D15	-0.4789	0.3814	-1.2556	0.2092
D17	-0.0028	0.0075	-0.3798	0.7041
D18	-0.0391	0.0152	-2.5806	0.0099

Log-Likelihood=-165.7520

Table 8.10.: Fitted Logistic Regression Model of FL Effectiveness on Dynamic Metrics - Reduced Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	1.2823	0.4619	2.7763	0.0055
D4	0.5247	0.2515	2.0859	0.0370
D7	-0.0181	0.0159	-1.1434	0.2529
D10	-0.1832	0.0503	-3.6455	2.6688e-04
D18	-0.0169	0.0061	-2.7907	0.0053

Log-Likelihood=-171.0955

T1, T6, T7, T9, T10, T12, and T13 are influential. However, metrics T12, T9, T10, and T6 (% Methods Covered in Failing Tests, hit Spectra Sparsity, % Method Coverage, and Diversity) have the greatest impacts (8.11).

Table 8.11.: Fitted Logistic Regression Model of FL Effectiveness on Test Metrics - Full Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	3.1581	2.0615	1.5320	0.1255
T1	-1.5633e-04	8.5626e-05	-1.8257	0.0679
T6	-3.5986	2.1508	-1.6732	0.0943
T7	0.0512	0.1046	0.4900	0.6242
T9	4.4552	2.4041	1.8532	0.0639
T10	3.9985	0.8760	4.5647	5.0033e-06
T12	-8.2022	1.8451	-4.4455	8.7703e-06
T13	-0.0032	6.0115e-04	-5.2836	1.2667e-07

Log-Likelihood=-165.7493

Bug. The fitted Regression model on Bug metrics is shown in Table 8.13. Since $LRT = 124.4032$ and $P[\chi^2(2) > 124.4032] < 0.00001$, one or more of the coefficients are nonzero. The reduced model is shown in Table 8.14. The comparison between the full and reduced models shows $LRT = 9.1822$ with $P[\chi^2(5) > 9.1822] = 0.102014$. Since the p-Value is large, exceeding 0.05, we conclude that the full model is no better than the reduced model. In fact,

Table 8.12.: Fitted Logistic Regression Model of FL Effectiveness on Test Metrics - Reduced Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	-0.1683	0.3699	-0.4551	0.6491
T1	-1.2424e-04	7.6622e-05	-1.6215	0.1049
T9	5.4445	2.3595	2.3075	0.0210
T10	3.7501	0.7808	4.8030	1.5632e-06
T12	-8.1328	1.7969	-4.5260	6.011e-06
T13	-0.0037	5.4135e-04	-6.8130	9.5552e-12

Log-Likelihood=-168.9407

there is little statistical justification for including the excluded metrics in the model. Metrics B2, B10, B12, and B14 are the influential metrics. Metrics B2 and B10 (# Classes Changed in the Patch and Avg. In-Degree of Faulty Node) have the greatest impacts (Table 8.14). It is noticeable that the metrics related to the location of bugs have more impact than the metrics related to the type of bugs. This confirms the finding by Diguseppe and Jones [39] that there is not any observable correlation between bug type and the FL effectiveness.

Table 8.13.: Fitted Logistic Regression Model of FL Effectiveness on Bug Metrics - Full Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	2.5360	0.5609	4.5207	6.1630e-06
B2	-0.9182	0.4576	-2.0063	0.0448
B9	0.0568	0.0474	1.1970	0.2312
B10	-0.5983	0.2623	-2.2802	0.0225
B12	-0.2238	0.0553	-4.0414	5.3121e-05
B14	0.0870	0.0274	3.1730	0.0015
B16	-0.0338	0.0542	-0.6239	0.5326
B19	-0.5542	0.3297	-1.6806	0.0928
B21	-0.6321	0.4993	-1.2658	0.2055
B22	0.1461	0.4054	0.3604	0.71851

Log-Likelihood=-171.1837

Table 8.14.: Fitted Logistic Regression Model of FL Effectiveness on Bug Metrics - Reduced Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	2.6456	0.5437	4.8662	1.1378e-06
B2	-1.0500	0.42525	-2.3207	0.0203
B10	-0.6459	0.2542	-2.5409	0.0111
B12	-0.1853	0.0336	-5.5166	3.4558e-08
B14	0.0768	0.0247	3.1091	0.0019

Log-Likelihood=-175.7748

StatDynaTest. Now, we can combine Static, Dynamic, and Test metrics in one model to check if a combination leads to a better fitted model. We do not add Bug metric, because Bug metrics are not known a priori and cannot be used for the prediction of FL effectiveness.

Table 8.15 shows the respective fitted model. With $LRT = 196.8849$ and $P[\chi^2(22) > 196.8849] < 0.00001$, we conclude that at least one or more of coefficients are nonzero. The reduced model is shown in Table 8.16. Again, $LRT = 22.8837$ with $P[\chi^2(12) > 22.8837] = 0.028736$ shows that the full model is slightly better.

However, LRT alone is not enough for comparing different models. We need more metrics to evaluate the goodness of fit.

Table 8.15.: Fitted Logistic Regression Model of FL Effectiveness on Stat-Dyna-Test Metrics - Full Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	0.6090	8.0583	0.0755	0.9397
S1	1117.7015	407.2250	2.7446	0.0060
S4	49.8753	17.2623	2.8892	0.0038
S5	-0.0004	0.0002	-1.9694	0.0489
S6	-0.0089	0.0112	-0.7993	0.4240
S10	-4.0124	4.7593	-0.8430	0.3991
S11	2.0062	1.2230	1.6403	0.1009
S13	0.5110	1.9381	0.2636	0.7920
S14	-2.6540	1.3486	-1.9679	0.0490
D4	1.54823	0.4034	3.8373	0.0001
D7	-0.0597	0.0287	-2.0750	0.0379
D9	0.0546	0.0310	1.7610	0.0782
D10	-0.2280	0.07173	-3.1786	0.0014
D15	0.8262	0.6042	1.36754	0.1714
D17	-0.0012	0.0083	-0.15113	0.8798
D18	-0.0434	0.0189	-2.2906	0.02198
T6	-5.3247	3.0852	-1.7258	0.0843
T7	0.0896	0.1316	0.6810	0.4958
T1	7.1405	0.0001	0.6159	0.5379
T9	2.7029	3.2701	0.8265	0.4084
T10	5.4856	1.3973	3.9256	0.0508
T12	-9.0923	2.5827	-3.5204	0.0004
T13	0.0020	0.0015	1.3404	0.1801

Log-Likelihood=-134.9428

Goodness of fit. Akaike Information Criterion(AIC) [3] is a measure of relative goodness of fit for a given set of data. Thus, it can be used for model selection. AIC is defined as:

$$AIC = -2 \times \text{LogLikelihood} + 2 \times (M + 1) \tag{8.4}$$

where M is the number of regression coefficients estimated for metrics. In general, lower values of AIC are preferred to larger ones. For example, between two models with 1000 and 2000 AIC values, the model with AIC=1000 is the better model. However, the AIC value 1000 on its own is meaningless and does not say anything about how well the model fits. AIC includes a penalty for complexity (the number of metrics). We use AIC to compare our generated models. Table 8.17 shows the results.

Table 8.16.: Fitted Logistic Regression Model of FL Effectiveness on Stat-Dyna-Test Metrics - Reduced Model

Metric	Coeff.	Std. Err.	tStat	p-Value
Constant	-3.2941	1.9287	-1.7078	0.0876
S1	683.0652	336.5434	2.0296	0.0423
S4	29.6761	11.6280	2.5521	0.0107
S5	-0.0003	0.0001	-2.1814	0.0291
S14	-0.4629	0.5927	-0.7810	0.4347
D4	1.2675	0.3266	3.8810	0.0001
D7	-0.0261	0.0194	-1.3436	0.1790
D10	-0.2324	0.0594	-3.9100	9.2290e-05
D18	-0.0444	0.01287	-3.4533	0.0005
T10	4.8228	0.8904	5.41608	6.0926e-08
T12	-5.5064	1.7324	-3.1784	0.0014

Log-Likelihood=-146.3847

The AIC results suggest that the model generated on the combination of Static, Dynamic and Test metrics is the best fitted model. Dynamic and Test models have better AIC scores than Static and Bug models. That is, Dynamic and Test metrics can better distinguish between “effective” and “ineffective” FL.

Table 8.17.: Comparing Fitted Regression Models on the Effectiveness of FL

Metric	AIC
Static-Full	400.0952
Static-Reduced	431.8978
Dynamic-Full	347.5040
Dynamic-Reduced	352.1910
Test-Full	347.4986
Test-Reduced	349.8814
StatDynaTest-Full	315.8857
StatDynaTest-Reduced	314.7694
Bug-Full	362.3674
Bug-Reduced	361.5496

Assessing the fit of model. Another way to evaluate a fitted Logistic regression is via a classification table. The performance of a model can be evaluated by comparing the predicted labels against the true labels of instances. One frequently used metric is *Accuracy* which is measured as:

$$Accuracy = \frac{\# \text{ of Correct Predictions}}{\text{Total } \# \text{ of Predictions}} \quad (8.5)$$

In our application, the coefficients produced by the model are used for predicting the label in a binary way. To have a binary prediction, we must define a cutpoint (C), and compare each estimated probability to C. If the estimated probability is larger than C then we let the predicted label to be equal to “effective”; otherwise it is equal to “ineffective”. The most widely used value for C is 0.5.

To evaluate a binary classification, Accuracy is not the most suitable metric. To avoid over-fitting and bias toward one label, we need to measure the performance of the models at various C values. For these cases, ROC (Receiver Operating Characteristic) curves and AUC (Area Under Curve) are better options [15]. ROC [47] is a probability curve, and AUC [47] is the area under ROC curve and represents the degree of separability. It measures how much model is capable of discriminating between classes. Higher the AUC, better the model is at distinguishing between “effective” FL and “ineffective” FL.

The ROC curve is plotted with True Positive Rate (TPR) on the y-axis against the False Positive Rate (FPR) on the x-axis.

$$TPR = \frac{TP}{TP + FN}, \quad (8.6)$$

$$FPR = \frac{TN}{TN + FP}, \quad (8.7)$$

where TP= # of positive instances predicted as positive, FN= # of positive instances predicted as negative, TN=# of negative instances predicted as negative, and FP=# of negative instances predicted as positive. An excellent model has AUC near to the 1 which means it has a good measure of separability. When AUC is 0.5, it means the model is not able to distinguish between classes.

Table 8.18 shows the AUC values for different models. To get a more complete evaluation, we included the results of five other classification techniques: Decision Tree [171], Support Vector Machine (SVM) [171], K-Nearest Neighbors (KNN) [171], Ensemble Boosted Trees [171], and Ensemble Bagged Trees [171]. All the results are based on 10-fold cross-validation.

Sturdivant et al. [152] mention that $0.7 \leq AUC < 0.8$ is considered as acceptable discrimination and $0.8 \leq AUC < 0.9$ is considered as excellent discrimination. SVM and Logistic Regression models have the best fitted models. The Logistic regression model on the reduced version of the combination of Static, Dynamic, and Test metrics (StatDynaTest-Reduced) has the highest AUC value (0.86) which is excellent discrimination. Based on AUC results, Dynamic and Test models are better fitted models than Static and Bug models.

8.5. Summary of Results

Now, we can answer the research questions and discuss the threats to validity.

8.5.1. Research Questions

RQ1: Based on regression analysis and classification results in Tables 8.17 and 8.18, the models generated on Dynamic and Test metrics are better fitted than the models generated on Static and Bug metrics. However, a full separation between different groups of metrics is not possible. The impact of tests on dynamic aspects of the code is inevitable.

Table 8.18.: Performance of Classifiers - Acc. (%)

Metric	Tree		Logistic Regression		SVM		KNN		Ensemble-Boosted		Ensemble-Bagged	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
Static-All	69.5	0.71	68.9	0.74	68.9	0.73	69.8	0.74	64.5	0.72	62.5	0.67
Static-Reduced	69.2	0.70	71.0	0.69	70.1	0.69	67.4	0.69	66.3	0.70	63.3	0.69
Dynamic-All	75.4	0.78	78.0	0.83	76.8	0.84	75.1	0.81	73.3	0.82	76.2	0.84
Dynamic-Reduced	75.4	0.78	75.7	0.83	77.1	0.83	75.1	0.83	75.1	0.82	77.4	0.84
Test-All	76.8	0.79	76.5	0.83	77.4	0.84	72.4	0.78	71.6	0.80	73.3	0.81
Test-Reduced	72.1	0.76	76.5	0.83	77.1	0.83	71.6	0.77	72.7	0.81	77.1	0.83
StatDynaTest-All	72.1	0.73	77.7	0.85	78.0	0.84	72.4	0.80	75.4	0.81	76.5	0.84
StatDynaTest-Reduced	74.5	0.75	79.8	0.86	79.5	0.84	73.0	0.81	77.1	0.80	76.2	0.82
Bug-All	72.4	0.71	75.4	0.80	73.9	0.80	69.5	0.79	71.8	0.77	71.3	0.76
Bug-Reduced	71.6	0.74	75.1	0.80	73.0	0.76	72.1	0.79	70.4	0.75	68.9	0.73

Also, as shown in Table 8.16, significant Dynamic metrics measure the degree which the call graphs are tangled. And Table 8.14 shows that the bug location is more important than the bug type. While the dynamic call graph of the tests are tangled and highly coupled, no matter where a bug happens it is difficult to localize it. Significant Static metrics too show the general complexity of the code. Nevertheless, based on Table 8.18 Static metrics are the least influential metrics.

A combination of Test, Dynamic, and Static gives the best fitted model with AUC=0.86 (see Table 8.18).

RQ2: In the following, we list the most significant features of each group along with the reason behind their influence. Since the StatDynaTest-Reduced is the best fitted model (Tables 8.18 and 8.17 and 8.16), we consider the metrics in this model as the most influential metrics. The general positive or negative impact of a metric on the effectiveness of FL is obtained from Table 8.6. The sign of the coefficients might change in combination with other coefficients.

Static

- S1 (# Methods with LoC>30): This metric has a negative impact on the effectiveness of FL. That is, as methods grow larger, the FL becomes more difficult. This metric has also been recognized as a simple criterion to estimate a component failure [127]. As methods grow to do more tasks, they will be executed in a larger subset of tests with different test goals. Having the same subset of methods executed in a large set of tests harms the localization effectiveness. Lippert and Roock [108] recommend that methods should not have more than an average of 30 LoC.
- S4 (% Methods with Nesting Depth>5): In addition to the LoC, as the level of nesting of statements, for example 'if', 'for', and 'while', in a method increases, the effectiveness of FL decreases. The rule of thumb suggests that the maximum nesting depth should be restricted to 5. As an example, assume a method with 2 nested if statements and 3 paths that have been assigned 3 tests, one for each path. If the execution of only one of the paths leads to failure, the method is executed more frequently in passing executions and less frequently in failing executions. Thus, it will not be considered highly suspicious. Therefore, the nested statements should be extracted into new methods [49].
- S5 (% Methods with $3 \leq \text{Nesting Depth} \leq 5$): Similar to S4.
- S14 (Mean # of Fields per Type): Having many fields is an indication that a class has grown too large. A class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain. We do SBFL at the method-level, and methods can take values as parameters and return a value as a result. Thus, we can imagine that the exceeding number of fields need enough number of various tests to decline ambiguity.

Dynamic:

- D4 (Mean Node Degree): An increase in the average number of incoming and outgoing calls of methods has a negative impact on the effectiveness of FL. This metric can be seen as a measure of coupling between methods. As a larger number of methods are connected to each other, a larger number of tests are covered in failing tests. Thus, the list of suspicious methods grows larger.
- D7 (Max Node Out Degree): Similar to D4.
- D10 (Graph Diameter): To find the diameter of a call-graph, one first finds the shortest path between each pair of methods. The greatest length of any of these paths is considered as the diameter of the graph. It is clear that increasing this value has a negative impact on the effectiveness of FL. That is, the large value of this metric means that usually there is a large number of methods executed between two method executions. Tests inevitably cover a large number of methods. Analogously to D4 and D7, this makes FL harder.
- D18 (Response for Class): This metric shows the interaction of the class' methods with other methods. That is, again, as this number increases, a larger number of methods are covered in tests. Thus, it can harm the effectiveness of FL. The RFC gives the count of all methods that can be invoked in response to a message to an object of the class or by some method in the class. The high value of RFC shows the high complexity of the class. If a larger number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated.

Test:

- T10 (% Method Coverage): % of method coverage has a positive impact on the effectiveness of FL. This metric considers both passing and failing tests. A higher coverage can add to the diversity in hit spectra. Therefore, it helps the FL.
- T12 (% Methods Covered in Failing Tests): Since all the methods in a failing test are somehow suspicious to be faulty, as the number of methods increases, the list of candidates for inspection grows larger.

RQ3. A Logistic regression model on the combination of metrics mentioned in RQ2 (Table 8.16) shows an excellent discrimination power. Thus, we propose to use it as a prediction model for the effectiveness of FL. Moreover, the best-fitted model shown in Table 8.16, can also be used to assign a *Confidence Factor* to the results of SBFL. This model is generated on 4 Static Metrics, 4 Dynamic metrics, and 2 Test metrics. Dynamic and Test metrics change in every debugging session. Thus, this model can be used a posteriori to decide whether to trust the SBFL results or not. However, to prove these claims, we need to evaluate them on unseen data sets. *Due to lack of proper data sets for testing purpose, we propose it as our outlook on the FL improvement.*

8.5.2. Threats to Validity

We set up our experiment on Defects4J, the most frequently used benchmark in the literature. Moreover, we used cross-validation technique to give our test and training setting more diversity. Nevertheless, we do not claim that our experimental setup and results will be valid for all other contexts. We explained our methodology to help other researchers to apply it in other contexts.

As an indicator of the effectiveness of FL, we utilized the effectiveness of SBFL techniques. We are aware that SBFL techniques are not representative of all different FL techniques. Nevertheless, they are widely used and studied and are considered as one of the most promising techniques [32]. As mentioned earlier, the selection of SBFL formula does not have any impact on the results.

Quality of the test suite has a great impact on the result of this study. As mentioned earlier the executed tests affect both Test and Dynamic metrics.

We do not claim that our metrics list is comprehensive. Initially, we collected 130 metrics. Later, we filtered them to exclude metrics that are not actionable.

8.6. Conclusion

Despite ongoing research, state-of-the-art FL techniques are not applicable in practice yet. In this chapter, we answered the question of what are the most influential metrics on the effectiveness of FL.

We believe that before continuing our endeavor in finding new techniques for FL, we need to understand why sometimes they are effective and sometimes not. The answer to this question can shift our focus from merely introducing new techniques to the quality of projects to facilitate fault localization.

Our analysis on 70 metrics on the Defects4J data set shows that 4 Static, 4 Dynamic, and 2 Test metrics are the most influential ones on the effectiveness of SBFL techniques. Developers should consider them to improve the quality of their code.

In addition, a model generated on these metrics can be used a priori to predict the effectiveness of FL. Thus, if the effectiveness probability is not high, developers should avoid using the FL tool and can refactor their code to facilitate FL. Also, this model can be used a posteriori to assign a confidence factor to the results of SBFL. A smart application of SBFL techniques, knowing where and when they would be helpful, avoids time and money waste and adds to their usability.

Similar to the previous chapter, the difficulty of gathering needed data for another case study prevents us from reporting on a second study. Thus, we propose the results of this chapter's case study as our outlook on the future of FL.

Part IV.

Related Work and Conclusion

9. Related Work

This chapter presents related work in the sub-fields of fault localization and failure clustering. Parts of this chapter have been published in peer-reviewed publications [55, 58, 54, 56, 150] co-authored by the author of this thesis.

A lot of research has gone into the fields of automated fault localization and failure clustering. As a result, we split this section into the following parts.

9.1. Case Study

The Siemens suite [70] is the first benchmark of buggy programs. It consists of 7 C projects with varying sizes between 141 and 512 LoC. All bugs are manually injected into the code.

Software-artifact infrastructure repository (SIR) [40] is another benchmark introducing reproducible bugs. SIR consists of 85 subjects written in Java, C, C++, and C#. The subject sizes vary from 24 to 503833 LoC. Most of the bugs introduced in SIR are artificial, either hand-seeded or obtained by mutation. It has only one C++ subject with 1034 LoC and artificial bugs.

CoREBench [13] is a collection of 70 regression bugs extracted from 4 C projects with more than 145 KLoC. ManyBugs [99] consists of 1183 defects extracted from 15 C programs with ca. 5900 KLoC.

iBugs project [27] is a database of real bugs for Java programs. It contains 3 subjects and 390 bugs extracted from version control history. 223 of the bugs are associated with failing tests. Nevertheless, the faulty versions can only be built with an outdated version of the JVM [81]. Defects4J [81] is a database of reproducible bugs for Java programs. It provides 357 bugs out of 5 subjects with 352 KLoC.

In contrast to Siemens and SIR, Pairika's subjects have between 10880 and 196550 LoC and all the bugs are real. In contrast to CoREBench, ManyBugs, Defects4J and iBugs, Pairika is a benchmark for C++ programs.

We believe that having realistic benchmarks for both Java and C++ programs is necessary for a realistic evaluation of our solutions. Thus, we used Pairika, LCM, and Defects4J for our evaluations.

9.2. Failure Clustering

Dickinson et al. [35] introduce the idea of clustering execution profiles to identify failing executions from passing executions. Podgurski et al. [148] utilize a supervised classification to group failing executions with similar causes. Bowering et al. [14] use Markov models to present program executions and then cluster the models to categorize the executions. Liu and Han [109] claim that trace similarity is not enough to cluster failing executions with respect to the causes. They propose the idea of using fault localization and suggest that two failing traces are similar if they roughly point to the same fault location. Jones et al. [77] utilize the previous ideas and combine the notions of clustering based on execution traces and clustering based on fault localization ranking to propose a new approach for clustering failures.

Jones et al. introduce two clustering techniques, the first of which we described in Chapter 4. As the second technique, they use fault localization results of each failing test and all passing tests and compute the pairwise similarity of the results. Then, clusters are formed based on similarities. In our approach in Chapter 4, we set up a training phase to adapt their first clustering technique to our context. As a result, our execution trace profiling and fault localization metric differ from their work. In addition, we have added a new mechanism to select some representatives for each cluster. In addition, our approach is methodologically different. We use this technique to segregate between failing tests rather than locating and repairing faults in parallel.

There is more literature that focuses on parallelizing fault localization and debugging [2, 167, 68] by using logic reasoning or integer linear programming. These studies only focus on segregating between fault to make them more localizable. However, they cannot cluster failing tests. On the other hand, in Chapters 4 and 5, we want to find failures that happened due to the same underlying faults rather than debugging and finding the underlying faults. Thus, the use case of our work is different.

Hsueh et al. [69] apply failure clustering in the context of graphical user interfaces. They instrument the code with the aid of developers that insert probing statements. They calculate the similarity and construct a tree. Then, considering the tree, they select some representative tests to start debugging. They suggest that just showing the tree to developers is beneficial. However, they do not decide about the cutting point for the clustering tree and do not explain clearly how they choose the representatives.

DiGiuseppe and Jones [37] use semantic concepts rather than the control-flow of the program to do failure clustering. They utilize latent-semantic-analysis to group failing executions based on their semantic intent. Their work differs from ours in their use of execution semantics as opposed to control-flow data. This approach would be beneficial when execution semantics or the used languages are not similar, which is not the case in our case study.

Rogstad and Briand [153] cluster deviations in regression testing for an industrial database system. Since it is unlikely to find a general solution for every context, they use context-specific profiles, namely test case specifications, and database manipulations, to

measure the similarity between failing executions. They use the Expectation Maximization algorithm [33] for clustering which requires a prediction for the maximum number of clusters. Their clustering method, feature sets, and application context differ from our work.

Recently, Pham et al. [146] grouped failing tests based on the similarity between symbolic execution paths, generated by the symbolic execution engine KLEE [19]. They argue that their symbolic analysis approach generates more fine-grained clusters that segregate different faults even if the call stacks of failures are the same. However, symbolic execution has scalability issues. For large industrial cases such as automotive companies, utilizing symbolic execution would be expensive.

Another related line of research involves clustering crash or bug reports [28, 51, 88, 125, 157, 111, 34]. Our focus is clustering failing tests rather than crash reports, bug reports or failing traces. These techniques explore run-time information collected in the field where the software systems are deployed.

Our second goal in Chapter 5, using our clustering tool a priori to select tests for the next regression test run, is somewhat related to test selection or prioritization area of research.

The aim of test prioritization [159, 189, 36, 188, 73, 45, 100, 191, 46, 48, 22, 41] is to minimizing test time and maximizing efficiency with running fault revealing tests first. Thus, these techniques rely on data such as statement coverage for fault detection and hope that satisfying these coverage goals will lead to an increasing fault detection rate. Most of these test case prioritization techniques are coverage based.

Regression test selection is a well-studied research topic [61, 52, 161, 63, 132, 186, 42, 16, 194, 53, 101, 192, 44, 122, 43]. These techniques compute test dependencies statically or dynamically to find the tests that are affected by the code changes. Rerunning only the affected tests reduces regression testing costs. Similarly, test-suite reduction helps in speeding up the regression test by eliminating the redundant tests [160, 12, 21, 23, 50, 60, 62, 72, 75, 106, 114, 190, 189, 197, 156, 155, 183, 184, 193]. All these techniques are based on the source code. However, we believe that utilizing our clustering technique, we will be able to add semantic similarity analysis to these approaches to improve their effectiveness. In their real world experiment report, Shi et al. [160] conclude that researchers should develop novel test suite reduction techniques that either miss fewer failed builds or at least provide more predictable fault detection loss.

To the best of our knowledge, our experiments explained in Chapters 4 and 5 are the first studies addressing the challenge of reduction of failure analysis time in the context of automotive CPSs for both SiL and HiL levels of testing. We are not aware of previous work that has utilized SiL-level execution profiles to analyze failing tests at the HiL level. The difference between SiL and HiL may be seen as a property specific to the domain of CPS. We do not think so. In fact, it is very similar to the difference between unit testing with mocks and testing of the integrated system with actual implementations. Thus, we believe that this idea is more general and can be applied in many domains.

9.3. Fault Localization

9.3.1. Spectrum-based Fault Localization

In their comprehensive overview of fault localization techniques [182], Wong et al. explored eight categories of fault localization, including spectrum-based techniques. Among these techniques were the three SBFL metrics that we considered during our evaluations (DStar2 [180], Ochiai [130], and Tarantula [78]).

Pearson et al. [142] performed a replication study on 10 claims in the literature regarding spectrum- and mutation-based fault localization. They refuted 3 of the claims on a data set of artificial bugs and refuted the rest of the results on the real bugs in the Defects4J database. Furthermore, they showed no significant relationship between results on artificial and real faults. Like Wong et al. [182], they too found DStar to be the best performing fault localization technique, however it was considered statistically indistinguishable from 4 other techniques, including Ochiai.

Keller et al. [85] evaluated the feasibility of applying 33 state-of-the-art SBFL metrics to a large real-world project, namely AspectJ. In a similar work, Tang et al. [172] published accuracy graphs of SBFL metrics. Also, De Souza et al. [30] surveyed SBFL techniques, their advances, and challenges.

Some of the studies have tried to improve the existing SBFL techniques by changing the hit spectra. Laghari et al. [96] proposed a variant of SBFL which extends the hit spectrum with the patterns of method calls extracted using frequent itemset mining. Zhang et al. [196] proposed PRFL which uses PageRank algorithm to recompute the spectrum information by considering the contributions of different tests. FLUCCS [166] uses existing SBFL formulae as features of the learning problem, instead of using raw spectrum data.

Test case optimization, prioritization and test suite balancing are other techniques to improve SBFL [105, 87, 11, 59], and etc. All these works show the importance of the test suite and also confirm our hypothesis that with limited code quality, even a good test suite achieves limited effectiveness.

Hit Spectrum Granularity

A technique for considering multiple levels of granularity when performing fault localization was proposed by Perez et al. [144]. However, their technique only considered one level of granularity at a time, starting at class level, and progressively narrowing the granularity down to the method and finally statement level for possible faulty components. In doing so, they saw an improvement in the instrumentation overhead required. Our work differs, in that we consider 18 different levels of program elements all at the same time. Additionally, our approach is not intended to be an improvement to instrumentation overhead, but rather an improvement to the overall fault localization results.

In another work [177], Wang et al. propose a multi-level similarity technique. Their technique selects useful test cases at the class level and computes code suspiciousness based

on comparing execution profiles at the block level. In our analysis, we also consider only relevant TCs at class level (a property is provided by Defects4J providers). However, our work differs, in that we consider 18 different levels when comparing the similarity not only the basic block.

In an effort to address some of the drawbacks of the method granularity, Sohn and Yoo [165] propose the use of Method-Level Aggregation, a technique that calculates the SBFL scores for statements and aggregates them up to the method-level by taking the highest scoring statement for each method. Their technique is designed to overcome the issue of methods on a single call-chain sharing the same spectrum values, resulting in tied SBFL scores, as well as the issue of some test cases only executing non-faulty parts of a faulty method. They were able to show that Method-Level Aggregation can improve the accuracy of existing techniques in some cases, but it cannot overcome the inherent limits of the given SBFL techniques. Our syntactic block granularity aims to address these same issues. However, it does so by considering finer-grained program elements in addition to just method-level elements.

While our work focused on different granularities of Executable Statement Hit Spectra (ESHS), there are other possible types of spectra that can be used with SBFL techniques. Harrold et al. did an empirical investigation of the differences in types of spectra, where they found that Complete Path (CPS), Path-Count (PCS), and Branch-Count (BCS) Spectra correlate well with fault localization [64]. Unlike ESHS, CPS records the complete path that was executed, PCS counts the number of times each intraprocedural, loop-free path was executed, and BCS counts the number of times each conditional branch was executed.

Faults of Omission

In their survey of fault localization techniques, Wong et al. also indicate that slice-based, as well as other fault localization techniques, struggle with localizing faults of omission [182]. However, the omission of code may still leave its mark on the program execution, such as the traversal of an incorrect branch. They argue that this implies that the developer is still able to identify suspicious code related to the omission error. Therefore, even though fault localization techniques may not be able to pinpoint the exact location of missing code, they can still provide a good starting point for where to look.

Along these lines, Lin et al. did a comprehensive study on the faults of omission in the Defects4J data set [107]. Of the 237 bugs they considered, they found that 110 included faults of omission. Of these bugs, they found that faults of data omission were harder to be localized than faults of control omission. Based on their findings as to the nature of the faults of omission in the Defects4J data set, they built a neural network model based on dynamic slicing which helped improve the accuracy in localizing faults of omission.

9.3.2. Fault localization with causal inference

In the first paper on utilizing causal inference for FL, Baah et al. [7] introduced a methodology to combine statistical techniques for counterfactual inference with causal graphical models to obtain causal-effect estimates that are not subject to severe confounding bias. Based on Baah's work, Shu et al. [162] applied causal inference methodology at the method-level. Their technique also incorporates a new algorithm for selecting covariates to use in adjusting for confounding bias. Later, Bai et al. [9] presented NUMFL, a value-based causal inference technique for localizing faults in numerical software. We proposed a new way to tackle confounding bias. To be able to compare the results, we implemented and improved MFL.

Call/data-dependency/control flow graphs

Call- and data-dependency graphs have been already used for debugging in different studies. Ko and Myers used call graphs in their works, [91] and [92], to describe the program behavior and to answer why and why not questions for a better debugging experience. Zhang et al. [196] used call graphs to find the connection between source methods in order to generate a new spectrum. Besides, as mentioned earlier, causality-based techniques such as MFL [162] used call- and data-dependency graphs to generate causal trees.

Liu et al. [110] used program dependence graphs to measure the accuracy of their localization technique. Henderson and Podgurski [67] used dynamic control flow graphs to find suspicious blocks of code. In our approach, we utilized call- and data-dependency graphs to find the relation between the faulty and suspicious methods to propose a re-ranking strategy for SBFL results and to assign a confidence factor to these results.

9.4. Failure Diagnosis Framework

To the best of our knowledge, there is no other tool that supports failure clustering and fault localization in one package.

Data generation. There is plenty of code coverage, instrumenting and profiling tools. Most code coverage tools cater to Java, followed by C, and C++, and some .NET. Among the open source tools for C++ code coverage, we picked OpenCPPCoverage.

Nevertheless, these tools do not generate a hit spectrum based on the coverage information. Gzoltar [20] is an Eclipse plug-in for testing and debugging Java programs which has the hit spectra generation feature. It is also compatible with the JUnit test framework ¹. Aletheia is a Visual Studio plug-in and is compatible with Google testing framework. We do not know any other openly accessible tool compatible with Google Test for generating hit spectra.

¹<https://github.com/junit-team>

Failure clustering. As mentioned earlier, there are several works on clustering failing (and passing) executions in the literature. Nevertheless, we are not aware of any openly accessible tool. Aletheia's failure clustering component receives a hit spectrum as input. Since the hit spectrum's representation is a simple CSV file, it is not limited to any specific programming language. In addition, it can select some representatives for each cluster to start the debugging process.

Fault localization. Pearson et al. [182] give a list of tools used in fault localization studies. There are different strategies for localizing faults in the literature. BARINEL [2] is a framework to combine spectrum-based fault localization and model-based diagnosis. GZoltar [20] is an automated testing and debugging framework using Ochiai metric. HSFal [80] is a slice spectrum fault locator. Pinpoint [1] is a fault localization tool using Jaccard coefficient. Tarantula [79] is a fault localization tool using Tarantula metric. Zoltar [71] is a spectrum-based fault localization tool. Zoltar-M [2] is a tool for detecting multiple bugs. Aletheia's fault localization component covers all the metrics implemented in tools mentioned above. Besides, it covers the DStar metric.

9.5. Predicting the Quality of Fault Localization

To the best of our knowledge, [97] and its descendant [98] are the only other papers on the prediction fault localization effectiveness. In their work, Le et al. introduced an oracle to predict whether the output of a fault localization tool is trustworthy or not. To build their oracle, they extracted some metrics based on the hit spectrum and fault localization scores which are somewhat similar to our Test metrics. However, in our approach, we collected Static, Dynamic, Test, and Bug metrics to have a more accurate model that can also be used a priori.

Our approach in analyzing the collected metrics is similar to the approach used in [10] for evaluating the impact of object-oriented design metrics on the quality of code. Basili et al. introduced some metrics that can be used to predict defects in the code. Software defect prediction is a well-studied and still active research area in the literature (e.g., [103, 104, 84, 66, 149]). Commonly used features in this area can be categorized into static metrics [123, 126, 74] and process metrics [170, 128, 176, 89]. The main difference between our paper and these papers is in the studied metrics. In addition to the static metrics, we worked on Dynamic, Bug, and Test suite related metrics.

We utilized SBFL techniques to drive our target labels. There is a plethora of studies on SBFL techniques in the literature (e.g., [196, 31, 24, 174, 178]). De Souza et al. in [32] and Pearson et al. in [141] reviewed and evaluated the most frequently used techniques.

10. Conclusions

This chapter first presents a summary of what has been done throughout the chapters of this thesis. Subsequently, we state the results of the thesis and the lessons learned during the development of this work. Afterward, we discuss limitations and avenues for future work.

As discussed in previous chapters, there are two types of techniques that help in reducing failure diagnosis time, failure clustering methods to group failing tests with respect to hypothesized faults and automated fault localization techniques to locate the faults.

10.1. Aletheia: Failure Clustering and Fault Localization in a Pipeline

Aletheia implements both of the above techniques to assist developers and testers to reduce failure diagnosis time. Where there are several failing tests due to multiple underlying faults, Aletheia helps users generate relevant data for further analysis while running the tests. After running the tests, it extracts failing tests and clusters them based on their underlying hypothesized faults. It also selects one (or more) representative tests for each cluster. Users can analyze representative failing tests to find all the faults without having to analyze all the failing tests one-by-one. Finally, if users need more help to locate the fault(s) in the code, it applies SBFL on each of the clusters to provide users a ranked list of program elements based on their suspiciousness to be faulty. Users can analyze the most suspicious elements first to find the fault(s). *However, only some parts of the ideas discussed in this thesis are implemented in Aletheia.*

Figure 10.1 shows Aletheia's main components: data generation (only for C++ programs), failure clustering, and fault localization. Each component can be used separately or the output of each one can be used as the input for the next one. Our tool can be used as a plug-in for Visual Studio or via command line. Aletheia is released as an open-source tool on Github: <https://github.com/tum-i22/Aletheia>, and a demo video can be found at: <https://youtu.be/BP9D68D02ZI>.

10.1.1. Data Generation

The first component intends to generate and prepare data for further analysis. To generate the data, Aletheia needs the source code and its test suite. Then, it runs test, collects

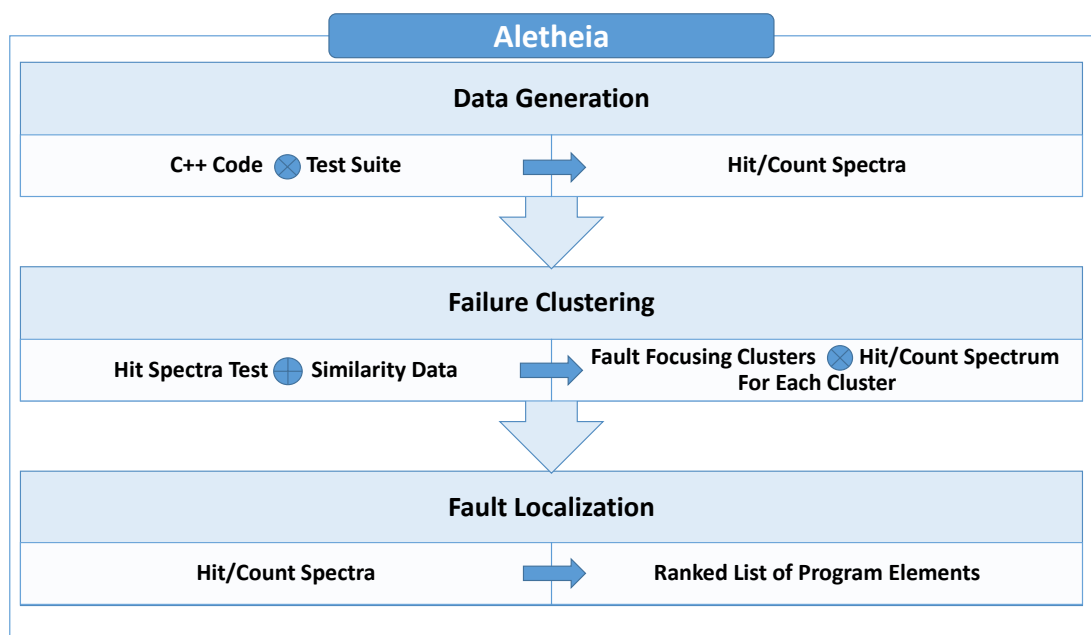


Figure 10.1.: Aletheia with Its 3 Main Components

coverage information and aggregates them into some hit/count spectra. To collect coverage information, it instruments the code using an open source tool, OpenCPPCoverage¹. OpenCPPCoverage measures code coverage for C and C++. Executing a program using this tool results in a report about which lines of code have been executed.

Aletheia first prepares test execution according to the start-up parameters of the given test suite. Then, it calls OpenCPPCoverage to execute tests while recording the coverage information and the test results. The final step is to generate a hit spectrum. Aletheia is able to generate 4 different kinds of spectra: **statement** (which statements have been executed), **function calls** (which functions have been invoked), **function calls with input parameters** (same as the previous one but separating calls considering different input parameters), and **function call count** (same as function calls, but recording count number of hits).

Since OpenCPPCoverage provides line coverage only, for other types of spectra, the final step is needed to partially parse the source files to aggregate the line-level information to provide statement- and function-level coverage.

Generated hit/count spectra are saved as csv files. Thus, they can be used as a data set for machine learning or as the input for failure clustering and/or fault localization.

The data generation component of Aletheia is designed for C++ code and is also com-

¹OpenCppCoverage, available at <https://github.com/OpenCppCoverage>, licensed under the GNU General Public License version 3 (GPLv3).

patible with Google testing framework². Google Test is a unit testing library for the C++ programming language, based on the xUnit architecture³.

10.1.2. Failure Clustering

Aletheia offers failure clustering as the second component. As input, it receives a hit spectrum (provided by the first component or by other tools such as GZoltar [20] for Java programs). It works based on the methodology explained in Chapter 4. First, it generates a hierarchical tree of failing tests (considering hit spectra as data set), then cuts the tree into some clusters utilizing SBFL. The number of clusters is decided based on the fault localization results. In the end, it selects k (user's desired number) representatives for each cluster. As output, users receive a csv file declaring the number of clusters, members of each cluster, and representative tests for each cluster, and also a csv file as a hit spectrum for each cluster (for further analysis).

10.1.3. Fault Localization

In Aletheia, four of the most popular metrics, Tarantula, Ochiai, Jaccard, and DStar (see Chapter 2) are implemented.

10.1.4. Evaluation

We already evaluated the failure clustering and fault localization ideas in the previous chapter. To complement our evaluations, we evaluate the whole framework considering two scenarios using Pairika and project Lang from Defects4J.

Scenario 1: In the first scenario, we use project Lang to evaluate our failure clustering and fault localization components in single and multiple fault cases. Among 65 faults introduced for Lang in Defects4J database, only 4 of them (Bug-30, -31, -34, and -57) induce more than two failing tests. We deliberately generate 3 cases for evaluation using these 4 bugs. In case 1, we include Lang-30, -31, and -39. In case 2, we include Lang-34. In case 3, we include Lang-57, -60, and -61. We include these combinations, in order to make the clustering/localization tasks more challenging. For each combination, we try to consider bugs that are in the same source file. We use Gzoltar to generate method level hit spectra.

The first option to reduce the failure analysis time is to cluster failures. Table 10.1 shows the evaluation results. The ARed metric is obtained from Chapter 2 and denotes the percentage of reduction in analysis time with respect to the best possible reduction. In some cases, Aletheia might find more than one cluster for one fault. Nevertheless, there is a huge reduction in analysis time. For example, in case 1, the 3 faults in the code induce 13 failures. Aletheia found 5 clusters. If we select one representative for each cluster, one

²<https://github.com/google/googletest>

³<https://xunit.github.io/>

should analyze only 5 failures to find all the faults rather than 33. This yields 80% reduction. One might get the list of representative tests and start the debugging process.

The second option is FL on found clusters in parallel. The segregation between failures, obtained from clustering, improves the accuracy of multiple fault localization. For example, Table 10.2 shows the ranks of faulty methods in case 3 before clustering and in found clusters. The results show that faults might mask each other. In this case, Lang-57 with 11 failures masks Lang-60 and -61 with 1 and 2 failures.

Scenario 2: In scenario 2, we use Pairika to evaluate our data generation and fault localization components. We did FL for 6 faults we extracted from Pairika. These faults are in Core, Photo, and Machine Learning modules with 196550, 10880, and 19398 LoC and 10528, 166, and 39 tests respectively.

Tables 10.4 and 10.3 shows the fault localization results using DStar and Ochiai metrics on 3 different spectra. As the results show, the faulty elements are not always ranked in top-5.

Table 10.1.: Clustering Effectiveness Using Average/Euclidean

	# of Failures	# of Faults	# of Clusters	ARed(%)
Case 1	13	3	5	80
Case 2	27	1	1	100
Case 3	14	3	3	100

Table 10.2.: Fault Localization Effectiveness Using DStar4

Spectrum	# of Bugs	Rank of Faulty Function(s)
Case 3	3 (bug-57,60,61)	1-3-8
Cluster 1	1 (bug-60)	1
Cluster 2	1 (bug-57)	1
Cluster 3	1 (bug-61)	1

Table 10.3.: Best Rank of Faulty Element Using DStar4 Metric

Component	Bug Id	Statement	Function Call	Function Call w. Input
ml2	#5413	43	4	28
ml3	#5911	1	3	240
photo1	#8706	1	1	98
photo2	#5045	1	1	117
core5	#8941	248	28	88
core14	#6380	1	300	715

Table 10.4.: Best Rank of Faulty Element Using Ochiai Metric

Component	Bug Id	Statement	Function Call	Function Call w. Input
ml2	#5413	43	4	28
ml3	#5911	1	3	240
photo1	#8706	1	1	98
photo2	#5045	1	1	117
core5	#8941	126	16	71
core14	#6380	1	38	114

10.2. Summary

This thesis presents a framework for failure diagnosis that can be used as an assistant tool for developers to reduce failure analysis time. The following is how developers and testers can benefit from using this framework:

Milica, a software tester at company X, is responsible for the quality of software development and deployment. She is involved in performing automated and manual tests to ensure the software created by developers is fit for purpose. Every week, she runs 1000 scheduled tests. Test runs take about 2 days. While running tests she collects coverage information (see Chapters 6 and 10). If not possible, she uses the database of similarity metrics between TCs (see Chapter 5).

After each test run, developers have 2 days to analyze failures, find bugs, repair them and mark the analyzed failed tests as ready for the next test run. Thus, Milica collects failing tests and clusters them (see Chapters 4 and 5). The clustering results include a prediction of the numbers of bugs, groups of failing tests that are failing because of the same reason, and one representative test for each cluster of failing tests. Milica, then, assign each group of failures to one developer and asks them to analyze only the representative failing tests to find the reasons behind failures asap.

Since locating the bugs in the code is very time consuming and tedious, Ana and Tabea, the developers, prefer to use automated fault localization tools and receive some hints about the location of the bugs. Ana uses the framework to predict the quality of fault localization on her piece of code (see Chapter 8). If the results are promising, she continues using the framework to do fault localization (see Chapter 7).

Tabea uses the prediction and does not receive promising results. She continues the debugging process without using the framework. However, later she uses the recommendations (see Chapters 7 and 8) to improve the quality of her code to facilitate fault localization.

Note: Automated failure diagnosis techniques can be very helpful in reducing failure analysis time regardless of the programming language. However, a minimum code and test quality is needed. If software is of poor quality, not only our framework but any other tool or technique cannot be effective on it. Thus, there is a prerequisite for using such tools and techniques (see Chapter 8).

10.3. Results and Lessons Learned

In this section, we highlight some of the most interesting results and discuss the lessons learned during the development of this thesis.

Clustering failures can reduce failure analysis time. Our results show that failure clustering is very effective in reducing failure analysis time. Clustering using coverage information reduces more than 80%, and clustering without coverage reduces more than 60% of analysis time.

Using syntactic blocks improves the effectiveness of SBFL. Our results show that syntactic block granularity has two advantages over existing granularities including statement and method level granularities. Syntactic blocks can help in reducing the wasted effort. Also, they give some hints about the type of the bug and can handle so-called missing faults.

However, the overall effectiveness of SBFL in an industrial setting depends on the quality of the code and test suite. The wasted effort even in case of using syntactic block granularity and re-ranking strategy might be higher than acceptable.

We can predict the effectiveness of SBFL. A model generated on 4 Static metrics (% of Methods with LoC>30, % of Methods with Nesting Depth>5, % of Methods with $3 \leq \text{Nesting Depth} \leq 5$, and Mean # of Fields per Type), 4 Dynamic metrics (Mean Node Degree, Max Node Out-Degree, Graph Diameter, and Response for Class), and 2 test metrics (% of Method Coverage and % of Methods Covered in Failing Tests) can be used as a predictor for the effectiveness of fault localization. In addition, developers can use these metrics to improve their code to facilitate fault localization.

The location of a bug is more important than its type. Our results show that what in the context of automated fault localization, the location of the bug is more important than its type. However, it again shows the importance of the quality of the code. Consider a piece of code with poor quality measures such as high coupling scores. Where ever a mistake occurs, it would be difficult to localize it automatically.

10.4. Limitations

The difficulty of gathering data for evaluation prevented us from reporting on several case studies. We evaluated our solution ideas on either C++ programs or Java programs. However, a comprehensive evaluation should contain benchmarks from both languages. Comparing the results, finding similarities and understanding dissimilarities, can help us better understand the important factors in general and in each category.

In our fault localization evaluations, we assumed that each buggy version contains only one bug. However, this is not the case in practice. Although we suggest that before any fault localization attempt, one should do clustering on failing tests to segregate between faults, a comprehensive evaluation should also consider cases with multiple bugs. We considered multiple bug cases in the evaluations of Section 10.1.

10.5. Future Work

In this section, we outline two avenues for future research:

Failure Clustering Regarding failure clustering, in our experiments, we used only coverage data or only non-coverage data. However, we believe that with combining these two, we can improve the fault detection rate of the reduced test suites. Thus, we plan to evaluate our approach considering a combination of non-coverage and coverage data for both failure clustering and test selection/prioritization.

In the future, we will consider more context specific data sources for failure clustering, for instance bus traces and log messages in car industry. Also, we plan to replace the current random weight optimization routine for feature sets with a genetic algorithm. The genetic algorithm can find the most promising combination of weights.

In addition, we will evaluate the effectiveness of our tool in scheduling and prioritizing TCs. Since tests which are in the same cluster tend to fail due to the same reason, selecting a subset of tests extracted from different clusters may help in finding the same amount of faults using less testing resources.

Fault Localization In the previous chapters, we saw that regardless of the FL technique used, it is not clear for developers whether the results are trustworthy or not. Even a highly effective FL technique might provide results or hints that are skewed due to different reasons (see Chapter 7). Thus, developers need an indicator that tells them to which extent they can trust each sessions' results. This confidence factor can help users to decide whether to continue looking into the ranking list or to ignore it. As future work, we look for a strategy to assign a **Confidence Factor** to the results of SBFL techniques. Our primary experiment in this regard is the following.

In Chapter 7, we introduced a re-ranking approach based on some relations between methods in a call graph. In addition to re-ranking, utilizing these relations, we define some rules to assign a confidence factor to the results of SBFL. Confidence factor estimates the probability of the fault being in the *Top-10* positions. For instance, "if the parents of the most suspicious element are ranked in *Top-5*, most probably one of the methods in *Top-5* is faulty". Thus:

The relative frequency of the cases where these rules hold and the fault is in the Top-10 is the heuristic which we use as the confidence factor.

Although confidence factor and the prediction model described in Chapter 8 are highly correlated, they are different. The prediction model predicts the effectiveness of SBFL on the program under consideration. However, a confidence factor shows how much one can trust the SBFL results in each debugging session. This may vary depending on the bug location. Using the prediction model, developers can decide whether they can use SBFL on a specific program. Using confidence factors, developers can determine whether they can trust the ranking list in each debugging session.

As shown in Chapter 7, the best performing combinations of Range and Mode parameters are Ancestor and Chain at Range 2. It means that considering parents, children, grandparents relations can be more helpful than looking into the ranking lists in a linear fashion. This leads us to derive 27 rules shown in Table 10.5 to measure the confidence of the ranking list.

The values in the table show the relative frequency of the fault being in among Top-10 positions if the condition of the rule holds. For example, considering project Chart, if the parent of the most suspicious method is in T-1 (*Parent in T-1*), in 66.66% of the cases, the fault is in this parent-child relationship. We can use this rule as a rule to assign a confidence factor. Thus, we translate it into “if the parent of the most suspicious element is in T-1, with 66.66% of probability the fault is there”. Given this number, developers can decide whether to consider or ignore the ranking list. In another example, if the parent and the child of the most suspicious method are in Top-10 (*Parent in T-10 & Child in T-10*), with 71.74% probability the fault is there.

The last column shows the coverage percentage for each rule. 19.52% coverage for rule 1 means that in 19.52% of the reviewed ranking lists Parent of the most suspicious element is also in T-1 (**Note:** As mentioned earlier, in SBFL, several elements can get the same rank due to different reasons such as having the same coverage profile. Thus, it is possible that several elements get rank 1).

Overall, the rules cover **88.84%** of the cases. If none of the rules hold in a ranking list, it does not mean that the ranking list is not helpful or is inaccurate. It only means it is not clear how good or bad it is. We are aware that despite 88.84% coverage, this list is not comprehensive, and external validity is a concern. However, we believe it sheds new light on how to use SBFL results.

As the table shows, some rules are stronger than others. Also, it shows that Closure has the least confidence values, analogously any FL technique is the least successful in Closure (see Chapters 7 and 6).

We deliberately consider 50% confidence factor as our threshold. Thus, We extract the rules with ≥ 50 in all projects as our final set of confidence rules. These rules are highlighted in Table 10.5. The interesting point is that all these strongest rules are considering upward relations like Ancestor-Range2.

Table 10.5.: Confidence Factors. The Relations are with Respect to the Most Suspicious Method in the SBFL Ranking List.

#	Rule	Chart	Lang	Time	Closure	Coverage(%)
1	Parent in T-1	66.66	88.88	71.42	37.03	19.52
2	Parent in T-5	66.66	90	70	50	32.27
3	Parent in T-10	75	86.36	76.92	51.06	37.45
4	Grandparent in T-1	33.33	-	-	55.55	47.8
5	Grandparent in T-10	66.66	60	100	50	13.15
6	No child	100	86.95	83.33	31.81	34.26
7	Child in T-5	69.23	91.89	66.66	40	43.82
8	Child in T-10	73.33	92.30	62.5	40	47.81
9	Infinity as suspiciousness score	75	76.47	66.66	25	12.75
10	Parent in T-1 & Grandparent in T-1	33.33	-	-	55.55	4.78
11	Parent in T-1 Grandparent in T-1	66.66	88.88	71.42	37.03	19.52
12	Parent in T-5 & Grandparent in T-10	50	60	100	50	9.96
13	Parent in T-5 Grandparent in T-10	72.72	90	75	50	35.46
14	Parent in T-10 & Grandparent in T-10	66.66	60	100	50	11.55
15	Parent in T-10 Grandparent in T-10	75	86.36	80	51.02	39.04
16	Parent in T-5 & Child in T-5	60	90.90	70	42.85	21.51
17	Parent in T-5 Child in T-5	70.58	91.3	66.66	45.76	54.58
18	Parent in T-10 & Child in T-10	71.42	90.90	72.72	45.16	23.90
19	Parent in T-10 Child in T-10	75	90	66.66	45.45	61.35
20	Parent in T-5 & no Child	100	88.88	66.66	50	9.16
21	Parent in T-5 no Child	78.57	88.23	81.25	40	57.37
22	Parent in T-10 & no Child	100	81.81	75	50	11.95
23	Parent in T-10 no Child	81.25	88.23	82.35	40.96	59.76
24	Parent & Grandparent are TestFun.	75	100	-	-	2.79
25	Parent Grandparent is TestFun.	87.5	89.65	66.66	50	47.41
26	Parent is TestFun. & no Child	100	92.30	100	66.66	9.16
27	Parent is TestFun. no Child	87.5	86.53	63.15	36.36	60.96

To evaluate these rules, we use project Math from Defects4J as our test data. The results are shown in Table 10.6. Each \checkmark means that the respective rule holds for the respective buggy version. The last column shows if the faulty method is ranked in Top-10. The last row shows the summary of the results. Rule 2 held in 5 cases. The last row shows that in 3 of these cases the fault is ranked in Top-10, and in the other 2 cases, the fault is not ranked in Top-10. The results suggest that rules 25, 26, 5, 20, and 22 are the best performing rules. However, this is a primary experiment, and the results are not generalizable.

The results also show none of the rules held in bugs -7, -23, and -30 where the fault is not ranked in Top-10. For bugs -24 and -28 several rules held but the faulty method was not listed in Top-10. We took a more in-depth look in these two versions. Considering version Math-24, 10 methods get rank 1. Since there are parent-child relationships between these 10 methods, most of the rules hold. Considering Math-28, the most suspicious method and its parents and children are 'Exceptions'. This bug is one example of the issue mentioned in

Chapter 7, regarding calling other methods after a failure. This issue skews the results from real fault to the methods called after the failure.

Table 10.6.: Evaluation of Confidence Rules on Math Project

Bug ID	Rule #										Top-10?	
	2	3	5	12	13	14	15	20	22	25		26
2										✓		True
4										✓		True
5										✓	✓	True
7												False
8												True
9										✓		True
10	✓	✓			✓		✓			✓		True
11										✓		True
13										✓		False
14										✓		False
17										✓		True
21	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	True
22										✓	✓	True
23												False
24	✓	✓	✓	✓	✓	✓	✓					False
25	✓	✓	✓	✓	✓	✓	✓			✓		True
26												True
27										✓		True
28	✓	✓			✓		✓					False
30												False
	T:3	T:3	T:3	T:2	T:3	T:2	T:3	T:1	T:1	T:11	T:3	
	F:2	F:2	F:0	F:1	F:2	F:1	F:2	F:0	F:0	F:2	F:0	

To sum up, it seems that the confidence factor analogously to SBFL is influenced by some factors. The results suggest that the confidence value is less in projects like Closure where SBFL has the least effectiveness. This is the second indication that quality of the code and test suite matter in the effectiveness of automated failure diagnosis techniques. Thus, we need to take into account the factors that influence the effectiveness of these techniques. We recommend this path as an essential future work since we believe it sheds new light on how to use SBFL.

Bibliography

- [1] R. Abreu, P. Zoetewey, and A. J. C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, 2007.
- [2] Rui Abreu, Peter Zoetewey, and Arjan J C Van Gemund. Localizing software faults simultaneously. *Proceedings - International Conference on Quality Software*, pages 367–376, 2009.
- [3] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, December 1974.
- [4] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the Accuracy of Fault Localization Techniques. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 76–87, 2009.
- [5] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411, 2005.
- [6] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 177–188, New York, NY, USA, 2016. ACM.
- [7] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, page 73, 2010.
- [8] GK Baah, A Podgurski, and MJ Harrold. Mitigating the confounding effects of program dependences for effective fault localization. ... *on Foundations of software ...*, pages 146–156, 2011.
- [9] Zhuofu Bai, Gang Shu, and Andy Podgurski. NUMFL: Localizing faults in numerical software using a value-based causal model. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.
- [10] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

- [11] Benoit Baudry, Franck Fleurey, Yves Le Traon, and Yves Le Traon. Improving Test Suites for Efficient Fault Localization. *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 82–91, 2006.
- [12] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings. 26th International Conference on Software Engineering*, pages 106–115, May 2004.
- [13] M. Böhme and A. Roychoudhury. CoREBench: studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 105–115, 2014.
- [14] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. *ACM SIGSOFT Software Engineering Notes*, 29(4):195, 2004.
- [15] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recogn.*, 30(7):1145–1159, July 1997.
- [16] L.C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Information and Software Technology*, 51(1):16 – 30, 2009. Special Section - Most Cited Articles in 2002 and Regular Research Papers.
- [17] Eckard Bringmann and Andreas Kr. Model-Based Testing of Automotive Systems. *2008 International Conference on Software Testing, Verification, and Validation*, pages 485–493, 2008.
- [18] Dekkers-Carol Bundschuh, Manfred. *Object-Oriented Metrics*, pages 241–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *USENIX*, 2008.
- [20] J. Campos, A. Ribeiro, A. Perez, and R. Abreu. GZoltar: an eclipse plug-in for testing and debugging. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012.
- [21] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie. How do assertions impact coverage-based test-suite reduction? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 418–423, March 2017.
- [22] Jinfu Chen, Lili Zhu, Tsong Yueh Chen, Dave Towey, Fei-Ching Kuo, Rubing Huang, and Yuchi Guo. Test case prioritization for object-oriented software. *J. Syst. Softw.*, 135(C):107–125, January 2018.

- [23] T.Y. Chen and M.F. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40(5):347 – 354, 1998.
- [24] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 184–191, Oct 2018.
- [25] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.
- [26] D. R. Cox. The regression analysis of binary sequences (with discussion). *Journal of the Royal Statistical Society*, page 215–242, 1958.
- [27] V. Dallmeier and Th. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 433, 2007.
- [28] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1084–1093, June 2012.
- [29] Jr. Stanley Lemeshow Rodney X. Sturdivant David W. Hosmer. *Applied Logistic Regression, Third Edition*. John Wiley Sons, Inc., 2013.
- [30] H. A. De Souza, M. L. Chaim, and F. Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.
- [31] Higor A. de Souza, Danilo Mutti, Marcos L. Chaim, and Fabio Kon. Contextualizing spectrum-based fault localization. *Information and Software Technology*, 94:245 – 261, 2018.
- [32] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.
- [33] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B*, 39(1):1–38, 1977.
- [34] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 333–342, Sep. 2011.
- [35] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on*

- Software Engineering*, ICSE '01, pages 339–348, Washington, DC, USA, 2001. IEEE Computer Society.
- [36] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. *SIGSOFT Softw. Eng. Notes*, 26(5):246–255, September 2001.
- [37] Nicholas DiGiuseppe and James A. Jones. Concept-based Failure Clustering. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 29:1–29:4, 2012.
- [38] Nicholas DiGiuseppe and James a. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering*, pages 928–967, 2014.
- [39] Nicholas Digiuseppe and James A. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Softw. Engg.*, 20(4):928–967, August 2015.
- [40] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. In *Empirical Software Engineering*, volume 10, pages 405–435, 2005.
- [41] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, New York, NY, USA, 2000. ACM.
- [42] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical evaluations of regression test selection techniques: A systematic review. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 22–31, New York, NY, USA, 2008. ACM.
- [43] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 11–20, New York, NY, USA, 2016. ACM.
- [44] Facebook. Buck. <https://buckbuild.com/>. Accessed: 2019-01-28.
- [45] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, Jun 2014.
- [46] ChunRong Fang, ZhenYu Chen, and BaoWen Xu. Comparing logic coverage criteria on test case prioritization. *Science China Information Sciences*, 55(12):2826–2840, Dec 2012.

- [47] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861 – 874, 2006. ROC Analysis in Pattern Recognition.
- [48] Yang Feng, Zhenyu Chen, James A. Jones, Chunrong Fang, and Baowen Xu. Test report prioritization to assist crowdsourced testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, 2015.
- [49] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [50] Jingyao Geng, Zheng Li, Ruilian Zhao, and Junxia Guo. Search based test suite minimization for fault detection and localization: A co-driven method. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering*, pages 34–48, Cham, 2016. Springer International Publishing.
- [51] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09*, pages 103–116, New York, NY, USA, 2009. ACM.
- [52] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 713–716, May 2015.
- [53] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 211–222, New York, NY, USA, 2015. ACM.
- [54] M. Golagha. A framework for failure diagnosis. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 561–562, March 2017.
- [55] M. Golagha and A. Pretschner. Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective. In *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2017*, 2017.
- [56] M. Golagha, A. M. Raisuddin, L. Mittag, D. Hellhake, and A. Pretschner. Aletheia: A failure diagnosis toolchain. In *To appear in: 2018 IEEE/ACM 40th International Conference on Software Engineering Companion*, 2018.
- [57] Mojdeh Golagha, Constantin Lehnhoff, Alexander Pretschner, and Hermann Ilmberger. Failure clustering without coverage. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, 2019.

- [58] Mojdeh Golagha, Alexander Pretschner, Dominik Fisch, and Roman Nagy. Reducing failure analysis time: An industrial evaluation. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP '17*, pages 293–302, Piscataway, NJ, USA, 2017. IEEE Press.
- [59] Alberto Gonzalez-Sanchez, Éric Piel, Rui Abreu, Hans Gerhard Gross, and Arjan J.C. Van Gemund. Prioritizing tests for fault localization. In *Situation Awareness with Systems of Systems*, volume 9781461462, pages 247–257. 2012.
- [60] Arnaud Gotlieb and Dusica Marijan. Flower: Optimal test suite reduction as a network maximum flow. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, pages 171–180, New York, NY, USA, 2014. ACM.
- [61] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*, pages 112–122, 2018.
- [62] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.
- [63] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01*, pages 312–326, New York, NY, USA, 2001. ACM.
- [64] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '98*, pages 83–90, New York, NY, USA, 1998. ACM.
- [65] Les Hatton. "characterising the diagnosis of software failure. *IEEE Software - SOFTWARE*, 07 2008.
- [66] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information Software Technology*, 59:170–190, 2015.
- [67] T. A. D. Henderson and A. Podgurski. Behavioral fault localization by sampling suspicious dynamic control flow subgraphs. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 93–104, April 2018.
- [68] Wolfgang Hogerle, Friedrich Steimann, and Marcus Frenkel. More debugging in parallel. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 133–143, 2014.

- [69] Chien Hsin Hsueh, Yung Pin Cheng, and Wei Cheng Pan. Intrusive test automation with failed test case clustering. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 89–96, 2011.
- [70] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. *Proceedings of 16th International Conference on Software Engineering*, (JANUARY 1994):191–200, 1994.
- [71] T. Janssen, A. Gemund, and R. Abreu. Zoltar: A Spectrum-based Fault Localization Tool. *Instrumentation*, 2009.
- [72] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, Feb 2007.
- [73] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, Nov 2009.
- [74] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 414–423, New York, NY, USA, 2014. ACM.
- [75] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.
- [76] J.a. James a Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering*, 2005.
- [77] James a. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in Parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07*, page 16, 2007.
- [78] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 273–282, New York, NY, USA, 2005. ACM.
- [79] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for fault localization. In *Proceedings of the Workshop on Software Visualization (SoftVis), 23rd International Conference on Software Engineering*, pages 71–75, May 2001.
- [80] X. Ju, X. Jiang, Sh.and Chen, X. Wang, Y. Zhang, and H. Cao. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, 90(1), 2014.

- [81] R. Just, D. Jalali, and M. D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 437–440, 2014.
- [82] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 654–665, 2014.
- [83] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 654–665, 2014.
- [84] Arvinder Kaur and Inderpreet Kaur. An empirical evaluation of classification algorithms for fault prediction in open source projects. *Journal of King Saud University - Computer and Information Sciences*, 30(1):2 – 17, 2018.
- [85] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. Van Hoorn, and D. Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*, 2017.
- [86] André I. Khuri. Introduction to Linear Regression Analysis, Fifth Edition by Douglas C. Montgomery, Elizabeth A. Peck, G. Geoffrey Vining. *International Statistical Review*, 2013.
- [87] Jeongho Kim, Jonghee Park, and Eunseok Lee. A new spectrum-based fault localization with the technique of test case optimization. *Journal of Information Science and Engineering*, 32(1):177–196, 2016.
- [88] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 486–493, June 2011.
- [89] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.
- [90] Klein Mitchel Kleinbaum, David G. *Introduction to Logistic Regression*, pages 1–38. Springer New York, New York, NY, 2002.
- [91] Andrew J. Ko and Brad A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 301, 2008.

- [92] Andrew J. Ko and Brad A. Myers. Extracting and answering why and why not questions about java program output. *ACM Trans. Softw. Eng. Methodol.*, 20(2):4:1–4:36, September 2010.
- [93] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 165–176, New York, NY, USA, 2016. ACM.
- [94] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 165–176, New York, NY, USA, 2016. ACM.
- [95] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 274–285, New York, NY, USA, 2016. ACM.
- [96] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 274–285, 2016.
- [97] Tien-Duy B. Le and David Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13*, pages 310–319, Washington, DC, USA, 2013. IEEE Computer Society.
- [98] Tien-Duy B. Le, David Lo, and Ferdian Thung. Should i follow this fault localization tool's output? *Empirical Software Engineering*, 20(5):1237–1274, Oct 2015.
- [99] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, Dec 2015.
- [100] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 510–514, Nov 2009.
- [101] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 583–594, New York, NY, USA, 2016. ACM.

- [102] Constantin Lehnhoff. Reducing failure analysis time: A data-driven approach. Master's thesis, Technical University of Munich, 1 2019.
- [103] J. Li, P. He, J. Zhu, and M. R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328, July 2017.
- [104] Libo Li, Stefan Lessmann, and Bart Baesens. Evaluating software defect prediction performance: an updated benchmarking study. *CoRR*, abs/1901.01726, 2019.
- [105] Ning Li, Rui Wang, Yu Li Tian, and Wei Zheng. An Effective Strategy to Build Up a Balanced Test Suite for Spectrum-Based Fault Localization. *Mathematical Problems in Engineering*, 2016, 2016.
- [106] Jun-Wei Lin and Chin-Yu Huang. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology*, 51(4):679 – 690, 2009.
- [107] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 509–519, New York, NY, USA, 2018. ACM.
- [108] Martin Lippert and Stephen Rook. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 1 edition, May 2006.
- [109] Chao Liu and Jiawei Han. Failure proximity: A fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14*, pages 46–56, New York, NY, USA, 2006. ACM.
- [110] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. SOBER: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, 2005.
- [111] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09*, pages 557–566, New York, NY, USA, 2009. ACM.
- [112] R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, Jun 1949.
- [113] L. Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

- [114] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. Test-suite reduction using genetic algorithm. In Jiannong Cao, Wolfgang Nejdl, and Ming Xu, editors, *Advanced Parallel Processing Technologies*, pages 253–262, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [115] Mehdi Maamar, Nadjib Lazaar, Samir Loudni, and Yahia Lebbah. Fault localization using itemset mining under constraints. *Automated Software Engineering*, 24(2):341–368, Jun 2017.
- [116] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook 2ed.* 2010.
- [117] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. 2008, 1(c):496, 2008.
- [118] Wes Masri. Automated Fault Localization. In *Advances in Computers*, volume 99, pages 103–156. 2015.
- [119] Wes Masri. Automated Fault Localization. Advances and Challenges. In *Advances in Computers*, volume 99, pages 103–156. 2015.
- [120] Wes Masri, Andy Podgurski, and David Leon. An empirical study of test case filtering techniques based on exercising information flows. 33:454–477, 08 2007.
- [121] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- [122] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP '17*, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press.
- [123] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, Dec 2010.
- [124] Sroka Michal, Nagy Roman, and Fisch Dominik. Specification-Based Testing Via Domain Specific Language. *Research Papers Faculty of Materials Science and Technology Slovak University of Technology*, 22(341):1–6, December 2014.
- [125] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. Automatically identifying known software problems. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 433–441, April 2007.
- [126] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190, May 2008.

- [127] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 452–461, New York, NY, USA, 2006. ACM.
- [128] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 382–391, May 2013.
- [129] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, page 342, 2011.
- [130] Akira Ochiai. Zoogeographical studies on the soleoid fishes found in japan and its neighbouring regions—ii. 22:526–530, 01 1957.
- [131] OpenCv. OpenCV Library, 2014.
- [132] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT '04/FSE-12*, pages 241–251, New York, NY, USA, 2004. ACM.
- [133] Tan Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*. 2006.
- [134] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.
- [135] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 199–209, New York, NY, USA, 2011. ACM.
- [136] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 199–209, New York, NY, USA, 2011. ACM.
- [137] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [138] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.
- [139] Judea Pearl. *Causality: Models, reasoning, and inference, second edition*. 2011.
- [140] Karl Pearson and Francis Galton. Vii. note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, 1895.

- [141] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering*, 2017.
- [142] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.
- [143] A. Perez, R. Abreu, and A. van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664, May 2017.
- [144] Alexandre Perez, Rui Abreu, and André Riboira. A dynamic code coverage approach to maximize fault localization efficiency. *J. Syst. Softw.*, 90:18–28, April 2014.
- [145] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, pages 654–664, Piscataway, NJ, USA, 2017. IEEE Press.
- [146] Van Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.
- [147] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun Jiayang Sun, and Bin Wang Bin Wang. Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003. Proceedings.*, 6:465–475, 2003.
- [148] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun Jiayang Sun, and Bin Wang Bin Wang. Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003. Proceedings.*, 6:465–475, 2003.
- [149] K. Punitha and S. Chitra. Software defect prediction using software metrics - a survey. In *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 555–558, Feb 2013.
- [150] M. R. Rahman, M. Golagha, and A. Pretschner. Poster: Pairika—a failure diagnosis benchmark for c++ programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 204–205, May 2018.
- [151] André Riboira and Rui Abreu. The gzoltar project: A graphical debugger interface. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques, TAIC PART'10*, pages 215–218, Berlin, Heidelberg, 2010. Springer-Verlag.

- [152] David W. Hosmer Jr. Rodney X. Sturdivant, Stanley Lemeshow. *Applied Logistic Regression, (3rd Edition)*. John Wiley Sons, Inc., 2013.
- [153] Erik Rogstad and Lionel C. Briand. Clustering Deviations for Black Box Regression Testing of Database Applications. *IEEE Transactions on Reliability*, 65(1):4–18, 2016.
- [154] Lior Rokach and Oded Maimon. Chapter 15— Clustering methods. *The Data Mining and Knowledge Discovery Handbook*, page 32, 2010.
- [155] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 130–140, May 2002.
- [156] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43, Nov 1998.
- [157] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 499–510, May 2007.
- [158] Sinan Saraçlı, Nurhan Dogan, and Ismet Dogan. Comparison of hierarchical cluster analysis methods by cophenetic correlation. *Journal of Inequalities and Applications*, 1(203):1–8, 2013.
- [159] Pavi Saraswat, Abhishek Singhal, and Abhay Bansal. A review of test case prioritization and optimization techniques. In M. N. Hoda, Naresh Chauhan, S. M. K. Quadri, and Praveen Ranjan Srivastava, editors, *Software Engineering*, pages 507–516, Singapore, 2019. Springer Singapore.
- [160] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. Evaluating test-suite reduction in real software evolution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*, pages 84–94, New York, NY, USA, 2018. ACM.
- [161] August Shi, Tiffany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 237–247, New York, NY, USA, 2015. ACM.
- [162] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. MFL: Method-level fault localization with causal inference. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pages 124–133, 2013.

- [163] Victor Sobreira, Thomas Durieux, Fernanda Madeiral Delfim, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *CoRR*, abs/1801.06393, 2018.
- [164] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*, 2018.
- [165] Jeongju Sohn and Shin Yoo. FlucCs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 273–283, New York, NY, USA, 2017. ACM.
- [166] Jeongju Sohn and Shin Yoo. FlucCs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 273–283, New York, NY, USA, 2017. ACM.
- [167] 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 - ISSTA 2013*, page 314, 2013.
- [168] Anja Struyf, Mia Hubert, and Peter Rousseeuw. Clustering in an object-oriented environment. *Journal of Statistical Software, Articles*, 1(4):1–30, 1997.
- [169] Barbara G. Tabachnick and Linda S. Fidell. *Using Multivariate Statistics (6th Edition)*. Pearson Education., Boston, 2013.
- [170] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15*, pages 99–108, Piscataway, NJ, USA, 2015. IEEE Press.
- [171] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
- [172] C. M. Tang, W. K. Chan, and Y. T. Yu. Theoretical, Weak and Strong Accuracy Graphs of Spectrum-Based Fault Localization Formulas. In *Proceedings - International Computer Software and Applications Conference*, volume 2, pages 78–83, 2017.
- [173] Ara C. Trembly. Software Bugs Cost Billions Annually. *National Underwriter / Life & Health Financial Services*, 106(31):43, 2002.
- [174] Jingxuan Tu, Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. On the analysis of spectrum based fault localization using hitting sets. *Journal of Systems and Software*, 147:106 – 123, 2019.

- [175] M. Utting, B. Legeard, and A. Pretschner. A Taxonomy of Model-Based Testing. *Software Testing, Verification and Reliability*, 22(April), 2006.
- [176] Shuai Wang, Shaukat Ali, Tao Yue, \Oyvind Bakkeli, and Marius Liaaen. Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search. *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 182–191, 2016.
- [177] X. Wang, Q. Gu, X. Zhang, X. Chen, and D. Chen. Fault localization based on multi-level similarity of execution traces. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 399–405, Dec 2009.
- [178] Y. Wang, Z. Huang, B. Fang, and Y. Li. Spectrum-based fault localization via enlarging non-fault region to improve fault absolute ranking. *IEEE Access*, 6:8925–8933, 2018.
- [179] E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 42–51, April 2008.
- [180] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014.
- [181] W. E. Wong, V. Debroy, Y. Li, and R. Gao. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 21–30, June 2012.
- [182] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, PP(99), 2016.
- [183] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *1995 17th International Conference on Software Engineering*, pages 41–41, April 1995.
- [184] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 522–528, Aug 1997.
- [185] 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, 2014.
- [186] G. Xu and A. Rountev. Regression test selection for aspectj software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 65–74, May 2007.

- [187] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52—63, Hong Kong, China, 2014. ACM.
- [188] S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou. A dynamic test cluster sampling strategy by leveraging execution spectra information. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 147–154, April 2010.
- [189] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.
- [190] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, pages 140–150, New York, NY, USA, 2007. ACM.
- [191] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09*, pages 201–212, New York, NY, USA, 2009. ACM.
- [192] L. Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209, May 2018.
- [193] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32, Sep. 2011.
- [194] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An empirical study of junit test-suite reduction. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering, ISSRE '11*, pages 170–179, Washington, DC, USA, 2011. IEEE Computer Society.
- [195] 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, ISSTA 2017*, pages 261–272, New York, NY, USA, 2017. ACM.
- [196] 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, ISSTA 2017*, pages 261–272, New York, NY, USA, 2017. ACM.
- [197] Hao Zhong, Lu Zhang, and Hong Mei. An experimental study of four typical test suite reduction techniques. *Inf. Softw. Technol.*, 50(6):534–546, May 2008.

List of Figures

1.1. A Hypothetical Faulty Method with Two Branches	6
1.2. The Framework's Flowchart - X is Predefined by Users.	11
1.3. Thesis Contributions	12
2.1. Hierarchical Clustering of 4 Failing Tests	16
3.1. TCF Actor	27
3.2. TCF Hil Mapping	27
3.3. TCF File - Minimal Working Example	28
4.1. Evaluation Results of Implemented Fault Localization Metrics	43
5.1. JSON Representation of Audio_Example.tcf	54
5.2. JSON Representation of Audio_Other_Example.tcf	55
5.3. Polynomial Regression Model for Cutting Distance	60
6.1. Faulty <i>mid()</i> Method. Each Syntactic Block is Enclosed by a Box	68
6.2. Faulty <i>mid()</i> Method, with Fault on Line 19. Each Syntactic Block is Enclosed by a Box	80
7.1. A Hypothetical Faulty Method with Two Branches	84
7.2. Human Patch to Fix Lang-26	86
7.3. An Example of Data-dependency Between Methods	87
7.4. Combined Call/Data-dependency Graph of Figure 7.3	87
7.5. Left Branch of Lang-26 Call Graph. Numbers in the Nodes Indicate the Line Numbers of Methods in the Source Code.	88
7.6. Human Patch to Fix Lang-9	89
7.7. Lang-9 Combined Call and Data-dependency Graph. A Data-dependency Edge Links the Faulty Node FastDate#116 to the Suspicious Node FastDate#180	90
7.8. Human Patch to Fix Chart-7	91
7.9. Chart-7 Combined Call and Data-dependency Graph.	92
7.10. Prevalent Patterns of Locality of Suspicious Method s^* to the Method that Contains a Fault e^*	93
7.11. Different Ranges; Steps Between the Most Suspicious and the Faulty Elements	93
7.12. Different Modes; Direction of the Path Between the Most Suspicious and the Faulty Elements	94

List of Figures

7.13. Triggering a Method After Test Failure	96
7.14. Lang-10. Calling testSdfAndFdp() After the Test Failure Makes It Wrongly the Most Suspicious Method	97
7.15. Combined Unit Tests	98
7.16. Combined Unit Tests	98
8.1. Study Data Set	105
10.1. Aletheia with Its 3 Main Components	130

List of Tables

2.1. A Hypothetical Hit Spectrum	21
2.2. Ochiai Suspiciousness Scores	22
3.1. Software Components of the Case Study	28
3.2. SiL Builds	28
3.3. HiL Builds [102]	29
3.4. Defects4J Projects [82] [31]	30
3.5. OpenCV Modules	31
4.1. List of Mutations Applied [68, 129, 4]	39
4.2. Average Values of Cophenet Correlation Coefficient on Training Data	39
4.3. List of Implemented Spectrum-Based Fault Localization Metrics [185, 113]	41
4.5. Number of Failures per Fault for Each Test Suite	45
4.4. Evaluation Results - Performance = 0.4 * FoundCauses + 0.2 * Purity + 0.4 * ARed	46
5.1. General Features [102]	52
5.2. Fail/Pass History - Example	52
5.3. Binary Failed History - Example	52
5.4. Binary Passed History - Example	53
5.5. Broken/Repaired History - Example	53
5.6. Jira History - Example	53
5.7. Binary Jira History	54
5.8. Binary TCF Similarity - Example	56
5.9. Performance Values Using Centroid Method [102]	57
5.10. Best Performing Weights for Input Feature Sets [102]	58
5.11. Evaluation Results - Performance = 0.4 * FoundCauses + 0.2 * Purity + 0.4 * ARed [102]	59
5.12. Average Scores of Performance Metrics Using Different Weights [102]	60
6.1. <i>mid()</i> Statement Granularity Ochiai Calculations	69
6.2. <i>mid()</i> Method Granularity Ochiai Calculations	69
6.3. <i>mid()</i> Syntactic Block Granularity Ochiai Calculations	70
6.4. Java Syntactic Block Types	71
6.5. RQ1 - No-omission Data Set Results	76
6.6. RQ1 - Full Data Set Results	76

6.7. RQ1 - No-omission Data Set Additional Metrics	77
6.8. RQ1 - Full Data Set Additional Metrics	77
6.9. RQ2 - No-omission Data Set Depth Metrics	78
6.10. RQ2 - Full Data Set Depth Metrics	79
7.1. RQ1 - Re-ranking Results Using Different Modes and Ranges. μ : mean EXAM, M: median EXAM, Top-10 (%)	95
7.2. RQ1 - Fault Localization Effectiveness - μ : mean EXAM, M: median EXAM, Top-10 (%)	95
7.3. RQ3 - Evaluation of Re-ranking Approach on Test Data Project Lang - μ : mean EXAM, M: median EXAM, Top-10 (%)	95
8.1. Static Metrics	102
8.2. Dynamic Metrics	103
8.3. Test suite Metrics	103
8.4. Bug Metrics	104
8.5. Correlated Metrics - (Correlation > 0.90)	106
8.6. Significance of the Coefficients - $P = P[X^2(1) > LRT]$	108
8.7. Fitted Logistic Regression Model of FL Effectiveness on Static Metrics - Full Model	109
8.8. Fitted Logistic Regression Model of FL Effectiveness on Static Metrics - Reduced Model	109
8.9. Fitted Logistic Regression Model of FL Effectiveness on Dynamic Metrics - Full Model	110
8.10. Fitted Logistic Regression Model of FL Effectiveness on Dynamic Metrics - Reduced Model	110
8.11. Fitted Logistic Regression Model of FL Effectiveness on Test Metrics - Full Model	110
8.12. Fitted Logistic Regression Model of FL Effectiveness on Test Metrics - Reduced Model	111
8.13. Fitted Logistic Regression Model of FL Effectiveness on Bug Metrics - Full Model	111
8.14. Fitted Logistic Regression Model of FL Effectiveness on Bug Metrics - Reduced Model	111
8.15. Fitted Logistic Regression Model of FL Effectiveness on Stat-Dyna-Test Metrics - Full Model	112
8.16. Fitted Logistic Regression Model of FL Effectiveness on Stat-Dyna-Test Metrics - Reduced Model	113
8.17. Comparing Fitted Regression Models on the Effectiveness of FL	113
8.18. Performance of Classifiers - Acc. (%)	115
10.1. Clustering Effectiveness Using Average/Euclidean	132

10.2. Fault Localization Effectiveness Using DStar4	132
10.3. Best Rank of Faulty Element Using DStar4 Metric	132
10.4. Best Rank of Faulty Element Using Ochiai Metric	133
10.5. Confidence Factors. The Relations are with Respect to the Most Suspicious Method in the SBFL Ranking List.	137
10.6. Evaluation of Confidence Rules on Math Project	138