Optimising and Assessing the Effectiveness of Spectrum Based Fault Localisation

Proefschrift voorgelegd tot het behalen van de
graad van Doctor in de Wetenschappen: Informatica
aan de Universiteit Antwerpen te verdedigen door:

Gulsher Laghari

Promotor
prof. dr. Serge Demeyer

Faculteit Wetenschappen
Departement Wiskunde-Informatica

Antwerpen, 2018

Universiteit Antwerpen

Cover illustration:
Fault localisation, which is finding the root cause of a failure, can be non-trivial and equivalent to *looking for a needle in a haystack*.

Optimising and Assessing the Effectiveness of Spectrum Based Fault Localisation

Gulsher Laghari

Promotor:

prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
Doctor in de wetenschappen: Informatica

This dissertation has been approved by

Promotor: prof. dr. Serge Demeyer

Doctoral jury

David Lo	Singapore Management University, Singapore
Rui Maranhao Abreu	University of Lisbon, Portugal
Bart Goethals	University of Antwerp, Belgium
Dirk Janssens	University of Antwerp, Belgium
Serge Demeyer	University of Antwerp, Belgium

To mother, Rabail, Zoha, Zaid, Rafay,…

Acknowledgments

My promotor Serge Demeyer kept telling me that the PhD journey is permanent state of identity crisis. This crisis finally comes to an end ....

Over the period of 4 years of this PhD journey, I firmly believe that this journey was not possible without the support of important people and you must acknowledge them.

First, I am grateful to Serge Demeyer for believing in me and providing me with the opportunity of his guidance and supervision. I particularly admire the academic freedom you allowed to follow and lead my own research, which has been giving me the courage, confidence, enthusiasm...

I would also like to thank Alessandro Murgia for helping me with my research. While sharing the office with you, I had the opportunity to have brainstorming sessions. This goes for Quinten Soetens as well.

I would particularly thank Martin Monperrus for discussing my research at Lille, pointing me to the dataset, and reviewing the early draft of ASE paper. This was really a great source of encouragement for me.

Next, I would like to thank my doctoral jury members Serge Demeyer, Bart Goethals, Dirk Janssens, David Lo, and Rui Maranhao Abreu for their feedback on this thesis and for the questions and interesting discussion at the preliminary defence. Special thanks to Rui and David for being part of my doctoral jury.

I thank to Ali, Brent, Diana for reviewing the early drafts of papers, dry runs for presentations, and research discussions. I thank all members of Ansymo Serge, Javier, Alessandro, Quinten, Ali, Diana, Dirk, Njima, Sten, Hans, Tim, Bart, Simon, Yentl, István, Claudio, and Joachim especially for joyful and fun discussions at the lunch time.

Similarly, I also thank to people who visited our lab and talked to me about my research including Thomas Fritz, Eleni Stroulia, and Marcelo de Almeida Maia.

I thank people at the secretariat including Caroline, Vera, and Mieke to help me with administrative work. Similarly, I would thank to Mrs. Lot Vanduffel for helping me with the accommodation at Antwerp. I thank many people at ITBS with whom I had a good

time. You all know I am referring to you so no need to enumerate your names :)

I am thankful to Siddiq, you received me at the Brussels airport when I arrived first time in Belgium and also provided me with company in initial days. Thanks to Altaf and Kirshan especially for travelling with me in Europe.

I owe to University of Sindh, Jamshoro who provided me with the necessary funding for whole PhD. Thanks also to people at University of Sindh for taking care of all the administrative and necessary work.

I also thank to Shahmurad, Saad, Kamran, and Yaqoob for always being with me through Whatsapp to ensure that I never miss the friendly chatter.

I would especially thank Abdul Razzaque to always support me and taking care of my family while I was away.

Thank you mom for always supporting and praying for me at every decision of my life. Your love, support, and prayers are my real strength in life. Finally, I thank to my family Rabail, Zoha, Zaid, and Rafay to complement my life and being with me.

Gulsher Laghari
Antwerp, May 2018

Abstract

Software, today, is the driving force in our modern society. However, the dynamic nature of the world has severe implications on the software as it must constantly adapt to ever changing requirements. This evolution then might introduce new faults or trigger dormant faults already lurking in the system. When the software is deployed with such faults, this may have unfortunate consequences.

Software testing acts as a safety net to detect these faults early on. Therefore, modern software teams spend lots of effort writing tests. Once the tests expose the faults, software developers need to fix them. The first —and the most difficult— step is to locate the exact location of the fault in the millions lines of code. Spectrum based fault localisation are techniques designed to aid developers in locating the fault. The main advantage of spectrum based fault localisation is that it only requires the faulty program and the set of test cases that expose the fault. With that input, the techniques statistically analyse the coverage information of the test cases and deduce a ranked list of possible locations. Unfortunately, in the current state of the art, spectrum based fault localisation have limited diagnostic accuracy: for some faults they succeed in pinpointing the exact location but for many others they miss.

This thesis aims to increase the effectiveness of spectrum based fault localisation. To this end, the thesis explores the use of *closed itemset mining* and *sequence mining* in spectrum based fault localisation and demonstrates that both *closed itemset mining* and *sequence mining* increase the effectiveness of spectrum based fault localisation. Moreover, we evaluate the spectrum based fault localisation from a new perspective; how does it perform on easy- and difficult-to-locate faults. We argue and demonstrate that defects exposed by component tests imply a larger search space and hence are difficult-to-locate compared to defects exposed by unit tests, which imply a rather smaller search space. We conclude that spectrum based fault localisation techniques perform far better on faults exposed by unit tests compared to faults exposed by component tests, hence there is still room to optimise it for difficult-to-locate faults exposed by component tests.

Nederlandstalige Samenvatting

Vandaag is software de drijvende kracht in onze snel evoluerende samenleving. De dynamiek van onze samenleving impliceert echter dat de software zich continu moet aanpassen of gedoemd is om uit te sterven. Dit aanpassingsproces introduceert onvermijdelijk fouten die soms ver strekkende gevolgen kunnen hebben.

Software tests fungeren als het eerste vangnet om fouten reeds in een vroeg stadium op te sporen. Zodra zo'n test een fout detecteert, moet ze gerepareerd worden. Om de fout te repareren moet eerst de oorzaak van de fout in de potentieel miljoenen regels beschikbare programmacode bepaalt worden. Spectrum based fault localisation is een speciale klasse van technieken ontworpen om programmeurs te helpen bij het vinden van de oorzaak van een fout. Deze technieken hebben enkele belangrijke voordelen, maar helaas is in de huidige stand van zaken de nauwkeurigheid beperkt: voor sommige fouten werkt het uitstekend, maar voor vele andere slaan ze de bal volledig mis.

Dit proefschrift heeft tot doel de effectiviteit van spectrum based fault localisation te verhogen. Hiervoor hebben we bestaande technieken gecombineerd met data-mining algoritmes (cfr., closed itemset mining en sequence mining) waarbij we vaststellen dat de nauwkeurigheid inderdaad verhoogd. Bovendien evalueren we die nauwkeurigheid vanuit een nieuw perspectief waarbij we een onderscheid maken tussen fouten blootgesteld via unit-tests (waar de zoekruimte beperkt is) en fouten blootgesteld door component-tests (waar de zoekruimte groter is). We stellen vast dat spectrum based fault localisation vooral goed werkt voor de eerste categorie.

Publications

Papers, directly or indirectly, included in this Thesis.

1. Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. **Localising Faults in Test Execution Traces**. In *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*, 1–8. Bergamo, Italy. August, 2015. URL: http://doi.acm.org/10.1145/2804360.2804361.

2. Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. **Improving Spectrum Based Fault Localisation Techniques**. In *Proceedings of the 14th Belgian-Netherlands Software Evolution Workshop (BENEVOL 2015)*, . Lille, France. December, 2015. URL: http://cristal.univ-lille.fr/evolille2015/program.html.

3. 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)*, 274–285. Singapore, Singapore. September, 2016. URL: https://doi.org/10.1145/2970276.2970308.

4. Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. **Fine-tuning Spectrum Based Fault Localisation with Sequence Mining**. In *Proceedings of the 15th Belgian-Netherlands Software Evolution Workshop (BENEVOL 2016)*, . Utrecht, the Netherlands. December, 2016. URL: https://benevol2016.wordpress.com/program/.

5. Gulsher Laghari and Serge Demeyer. **On the Use of Sequence Mining within Spectrum Based Fault Localisation**. In *Proceedings of the Symposium on Applied Computing (SAC 2018)*, 1916–1924. Pau, France. April, 2018. URL: https://doi.org/10.1145/3167132.3167337.

6. Gulsher Laghari and Serge Demeyer. **Poster — Unit Tests and Component Tests do Make a Difference on Fault Localisation Effectiveness**. In *Proceedings of the 40th*

International Conference on Software Engineering Companion (ICSE-C 2018), . Gothenburg, Sweden. May–June, 2018.
URL: https://www.icse2018.org/event/icse-2018-posters-poster-unit-tests-and-component-tests-do-make-a-difference-on-fault-localisation-effectiveness.

Other papers published/produced during the course of the PhD.

7. Gerardo Orellana, Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. **On the Differences between Unit and Integration Testing in the TravisTorrent Dataset**. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, 451–454. Buenos Aires, Argentina. May, 2017.
URL: https://doi.org/10.1109/MSR.2017.25.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our world and society are shaped and governed by software: almost all devices, machines, and artefacts surrounding us incorporate software to some extent. Additionally, the numerous organisations, businesses, and enterprises we encounter on a daily basis could not function without software—a world without software is unimaginable. However, these added benefits offered by software accompany the intense unintended sufferings and losses manifested via software failures [1, 2, 3]. Consequently, building software systems without faults is the *holy grail* of the software engineering community and lots of research has been conducted into efficient ways of finding faults at all stages of the software production process.

The advent of xUnit like automated testing frameworks made software testing easier [4]. Moreover, testing is the fundamental driver of *continuous integration*— an important and essential phase in a modern release engineering pipeline [5]. Thus, software testing ensures to filter out faults early on and serves as a safety net for fault detection—an oracle. Once the fault is detected through testing, the first step in *debugging* is fault localisation, that is to precisely pinpoint the faulty code.

To help developers quickly locate the faults, there exist automated fault localisation techniques. These techniques produce a ranked list of program elements indicating the likelihood of a program element causing the fault. User studies hint that they can positively aid developers in debugging [6]. These techniques can be seen as two broad categories, information retrieval based fault localisation and spectrum based fault localisation, based on the inputs required and the type of analysis done. Information retrieval based fault localisation uses bug reports and source code files for *static analysis* [7, 8, 9, 10], while spectrum based fault localisation uses program traces generated by *dynamic analysis* [11, 12, 13, 14]. Since spectrum based fault localisation techniques only require traces

from test runs— readily available after running the regression test suite— these heuristics are ideally suited for locating regression faults.

1.1 SPECTRUM BASED FAULT LOCALISATION

Spectrum based fault localisation is a lightweight yet quite effective heuristic to quickly pinpoint the faults. It requires as input the faulty program and the test suite, and produces as output a ranked list of program elements with the aim to place the faulty elements on top of the list. It runs the test suite on faulty program to collect the coverage of program elements. The coverage of each element is represented in a tuple of four values called hit-spectrum, which is used by fault locator functions as input to output the suspiciousness of the element indicating its likelihood to be faulty. These hit-spectra represent the abstract behaviour of the program and are the only means to reason about the fault, in this thesis we refer to this as raw spectrum analysis (See more details in Section 3.2). The raw spectrum analysis does not need any information as input other than the faulty program itself and the tests that detect the fault.

1.2 PROBLEM STATEMENT

Today, spectrum based fault localisation can be seen in a three dimensional space. The first dimension is *the granularity*, researchers have been experimenting with the different levels of granularity as program element (e.g. statements, blocks, methods, and classes). The second dimension is *the fault locator*, researchers have been experimenting with the different fault locator functions (e.g. from molecular biology [15], association measures [16], through a theoretical model [17], and genetic programming [18]). The third dimension, which is also the subject of this thesis, is *changing the hit-spectrum*. Other researchers did similar investigations recently, e.g. time-spectrum [19], method invariants [20], and code metrics [21]. See more in Section 3.2 and Section 4.6.

In this thesis, we explore the third dimension as well, *changing the hit-spectrum* via *closed itemset mining* and *sequence mining*. To verify the effectiveness of *closed itemset mining*, we first do a pilot study. As a pilot study, we replicate AMPLE, a tool which computes the sequence of method calls by sliding a window over the trace to locate the faulty classes by comparing the sequence of method calls between failing and passing tests [22]. Thus, we attempt to answer the preliminary research question.

> *Do the patterns of method calls extracted via closed itemset mining help boost the fault localisation accuracy in locating the faulty classes?*

To answer this research question, in Chapter 2 we create a new fault localisation

heuristic (SPEQTRA), which leverages *closed itemset mining* [23] to compute the sequence of method calls and compares these sequences to pinpoint faulty classes. We compare SPEQTRA against AMPLE on the NanoXML dataset and demonstrate the following.

> *SPEQTRA can immediately pinpoint the faulty class in 56% of all test runs, whereas with AMPLE it is only 40%. Moreover, for 70% of the faults, SPEQTRA has at most one false positive whereas for AMPLE this happens for 59% of the faults.*

Since the patterns of method calls extracted via *closed itemset mining* are effective in locating faulty classes, this gives the confidence to explore their application from course-grained granularity *classes* to fine-grained granularity *methods*. To this end, we modify the hit-spectrum with the patterns of method calls extracted via *closed itemset mining* to locate the faulty methods. Contrary to raw spectrum analysis where the hit-spectrum of a method indicates whether or not the method is involved in test cases, we modify the hit-spectrum with call patterns of the method. The hit-spectrum of call patterns for a method not only indicates whether or not the method is involved in a test case, but also summarises its run-time behaviour. In this thesis we refer to this spectrum analysis as patterned spectrum analysis (See details in Section 3.4.4). Like raw spectrum analysis, patterned spectrum analysis also does not need any information as input other than faulty program itself and the tests that detect the fault. Then, we attempt to answer the primary research question.

What is the overall performance of patterned spectrum analysis?

To answer this research question, we compare the two spectrum analyses (patterned spectrum analysis and raw spectrum analysis) experimentally on Defects4J dataset in Chapter 3 and find the following.

> *For 68% faults in the dataset, the patterned spectrum analysis performs better than raw spectrum analysis. The patterned spectrum analysis improves by 14% points in ranking the root cause of the fault in the top 10. Moreover, patterned spectrum analysis is more stable than raw spectrum analysis when the size of the ranked list increases.*

Since *closed itemset mining* misses the order of method calls and does not allow repetitive calls in the pattern, next we explore what the effect of *sequence mining* is. For this we replace the *closed itemset mining* with *sequence mining* in the algorithm presented in Chapter 3 and refer to it as sequenced spectrum analysis. Then we ask the primary research question.

What is the effectiveness and efficiency of sequenced spectrum analysis?

To answer this research question, we compare the sequenced spectrum analysis against raw spectrum analysis) experimentally on the Defects4J dataset in Chapter 4 and observe the following.

> *Comparatively sequenced spectrum analysis gains 12% improvement in ranking the faulty method at top and reduces the average wasted effort from 96.73 to 25.88 implying that on average ≈ 26 non-faulty methods need to be inspected in vain before pinpointing the faulty method. However, due to the additional overhead induced by the mining algorithm, sequenced spectrum analysis becomes impractical for large projects.*

Finally, we evaluate the effectiveness of spectrum based fault localisation by distinguishing between easy- and difficult-to-locate faults. We argue and demonstrate that defects exposed by component tests imply a larger search space and hence are difficult-to-locate compared to defects exposed by unit tests which imply a rather smaller search space. Thus, we ask the primary research question.

What is the performance of spectrum based fault localisation on easy-to-locate faults (exposed by unit tests) and difficult-to-locate faults (exposed by component tests)?

To answer this research question, we separate the faults in Defects4J dataset into two main categories (faults exposed by component tests and unit tests) in Chapter 5. Categorising the faults into easy- and difficult-to-locate faults based on the type of the test may not necessarily represent the optimum strategy. Yet, it lays out a foundation by providing a ballpark estimate on the effectiveness of spectrum based fault localisation on easy- and difficult-to-locate faults [24]. Then, we construct a suite of spectrum based fault localisation, attempt to tease out the performance difference of spectrum based fault localisation on easy- and difficult-to-locate faults, and demonstrate the following.

> *All spectrum based fault localisation techniques perform far better on faults exposed by unit tests compared to faults exposed by component tests, which confirms that the performance of a spectrum based fault localisation technique depends a lot on the presence of faults exposed by unit tests and component tests in the dataset.*

1.3 DATA MINING AND FAULT LOCALISATION

Several researchers have incorporated a plethora of data mining techniques to further improve software fault localisation. The intrinsic motivation for all of these attempts is that for a heuristic to be effective it needs to incorporate more information about the context of the fault. Data mining may achieve this by focussing on temporal relationships, exceptional behaviour, ...

In their pioneering work, Hsu et al. introduced the term *bug signature* to provide the developers with the context for fault localisation [25]. For example, when the statement s_1 is followed by the execution of statement s_2 results into the failure. They defined the bug signature as the sequences of program elements that, when executed in order, are likely to lead to a failure. In their work the program elements are statements corresponding to method entries and branches. The bug signature is mined from the traces and is a pattern representing the common subsequence.

Inspired by the work of Hsu et al., which encodes mining the bug signature as the problem of common subsequence identification, Cheng et al. define the problem as a graph mining problem [26]. Thus, they model execution traces as a graph and refer to this graph as software behaviour graph and mine the top-k discriminative subgraphs as bug signatures. Lo et al. optimised the top-k subgraphs as bug signatures to mine minimal signatures and signatures from disjoint graphs also [27]. This work has also lead to further optimisations as predicated bug signatures [28, 29].

Similarly, in this thesis, we also use mining algorithms— *closed itemset mining* and *sequence mining*— to mine patterns representing the execution summaries to optimise spectrum based fault localisation.

Mapping the terminology. In data mining a record of events is referred to as a *transaction* and the set of transactions as the *database*. The input to any mining algorithm is the database of those transactions along with the interestingness measure.

In this thesis we map the data mining terminology as follows. In Chapter 2, we are interested in the patterns of a class during the the execution of a test case. Thus, the transaction is the set of all outgoing method calls from an *object* of that class and the database comprises all the transactions for each created object of that *class*. Then, the call patterns for this class for the test case are mined from this database.

In Chapter 3, we are interested in the patterns of a method (thus more fine-grained than a class) during the the execution of a test case. Therefore, a transaction in that case is all the outgoing method calls from a single method with the same object instance during the execution of a test case. The database then comprises all the transactions from all different objects instantiated during a test execution. Then, the call patterns for the single

method for the test case are mined from this database.

1.4 CONTRIBUTIONS

The main contributions of this thesis are as follows.

- We replicate a study which uses sequence of method calls extracted via *closed itemset mining*, and demonstrate that *closed itemset mining* boosts spectrum based fault localisation in locating faulty classes.
- We propose the use of *closed itemset mining* in context of spectrum based fault localisation. Thus, we create a new spectrum based fault localisation technique by modifying the hit-spectrum with the patterns of method calls obtained via *closed itemset mining*. We demonstrate that *closed itemset mining* improves the effectiveness of spectrum based fault localisation.
- We apply *sequence mining* in context of spectrum based fault localisation by replacing *closed itemset mining* with *sequence mining* to measure and evaluate the effect of *sequence mining*. We also measure the performance of 47 fault locators with both raw spectrum analysis and sequenced spectrum analysis on the Defects4J dataset and rank them according to their performance.
- We refine the Defects4J dataset and separate the faults into two categories; faults exposed by unit tests and faults exposed by component tests. We show that the search space to locate the faults exposed by unit tests is smaller, hence these represent easy-to-locate faults and the search space is larger for faults exposed by component tests, hence represent difficult-to-locate faults.
- Finally, we assess the effectiveness of spectrum based fault localisation with a new evaluation perspective. We measure how spectrum based fault localisation techniques perform against faults exposed by unit tests and component tests, and show that the performance of spectrum based fault localisation techniques decreases on faults exposed by component tests confirming that these are indeed difficult-to-locate faults.

1.5 THESIS OUTLINE

The thesis is structured as follows. Chapter 2 explores and evaluates the use of *closed itemset mining* to locate the faulty classes. Chapter 3 studies the application of *closed itemset mining* further to locate the faulty methods. Chapter 4 plugs in the *sequence mining* in place of *closed itemset mining*, presented in Chapter 3, to measure the effect of *sequence mining* on the fault localisation effectiveness. Chapter 5 assesses the effectiveness of spec-

trum based fault localisation techniques on easy-to-locate faults exposed by unit tests and difficult-to-locate faults exposed by component tests. Finally, Chapter 6 recapitulates the thesis contributions and provides an outlook on future work.

1.6 ORIGIN OF CHAPTERS

Each of the chapters in the thesis is peer-reviewed. Chapters 2 to 4 were published in peer-reviewed software engineering venues while Chapter 5 is not published yet— only an extended abstract is published.

CHAPTER 2 was published in the *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)* [30].

CHAPTER 3 was published in the *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)* [31]. An earlier version of the chapter appeared in the *Proceedings of the 14th Belgian-Netherlands Software Evolution Workshop (BENEVOL 2015)*.

CHAPTER 4 was published in the *Proceedings of the Symposium on Applied Computing (SAC 2018)* [32] in SVT - Software Verification and Testing Track. An earlier version of the chapter appeared in the *Proceedings of the 15th Belgian-Netherlands Software Evolution Workshop (BENEVOL 2016)*.

CHAPTER 5 is not published yet. However, an earlier version of the chapter will appear as an extended abstract in the *Proceedings of the 40th International Conference on Software Engineering Companion (ICSE-C 2018)* [33].

Chapter 2

Localising Faults in Test Execution Traces

Localising Faults in Test Execution Traces

Gulsher Laghari, Alessandro Murgia, and Serge Demeyer

In *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*, 1–8. Bergamo, Italy. August, 2015.
DOI: http://doi.acm.org/10.1145/2804360.2804361.

This chapter was originally published in the *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*.

ABSTRACT

With the advent of agile processes and their emphasis on continuous integration, automated tests became the prominent driver of the development process. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In this chapter we propose a heuristic named SPEQTRA which mines the execution traces of a series of passing and failing tests, to localise the class which contains the fault. SPEQTRA produces ranking of classes that indicates the likelihood of classes to be at fault. We compare our spectrum based fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (NanoXML) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.

2.1 INTRODUCTION

The quintessential principle of continuous integration declares that software engineers should merge their working copies with the main branch several times a day [34].

During each integration step, a continuous integration server builds the entire project, using a fully automated process involving compilation, unit tests, integration tests, code analysis, security checks, etc. When one of these steps fails, the build is said to be *broken*; development can then only proceed when the fault is repaired [35, 36]. The safety net on automated tests, encourages software engineers to write lots of tests — several reports indicate that there is more test code than application code [37, 38, 39]. Moreover, executing all these tests sometimes take several hours [40]. Hence, it is critical to quickly identify the location of the fault in the code. Not only does a broken build block all progress in the team, but more importantly the location of the fault serves as an indicator for the software engineer expected to repair the build.

In the simplest case, there is a one-to-one mapping between the failing test and the class containing the fault. However, for complex object interactions where objects must adhere to a certain protocol, illegal call sequences trigger faults which are notoriously hard to pinpoint to an exact location [41]. Faults induced by illegal method call sequences are real and hard to debug: a conservative estimation identified 115 faults related to missing method calls in the Eclipse bug repository [42]. For such faults the one-to-one mapping between the failing test and the class containing the fault does not hold and then software engineers resort to debugging [43].

Luckily, there is a class of heuristics —named *spectrum based fault localisation*— which give indications for the location of the fault. Such heuristics compare execution traces of passing tests against the ones from a failing test, assuming that the points where the traces differ are the most likely location of the fault [13]. The state of the art heuristic for class level fault localisation by analysing method call sequences in unit tests is proposed by Dallmeier et al. with a tool called AMPLE [22]. AMPLE traces method calls invoked by objects and collects call sequences of the corresponding classes by sliding a window over the executions traces. It then compares the execution trace of all passing tests against one trace with a failing test and deduces which class most likely contains the fault. AMPLE was shown to be quite effective on a small yet representative case (NanoXML): it could immediately pinpoint the faulty class in 36% of all test runs; while on average 21% of the executed classes (10% of all classes) must be inspected to find the location of the fault.

In this chapter we report on a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. SPEQTRA addresses two shortcomings of the AMPLE heuristic: (a) it filters out repetitive method calls (e.g. contained in loops) and (b) avoids the sliding window and the arbitrary upper limit on the length of the call sequence it imposes. To address these shortcomings, SPEQTRA uses a different algorithm (closed itemset mining) to characterise method call sequence and distinguish the faulty ones. Via the replication experiment we demonstrate that the ranking of classes proposed by SPEQTRA is signifi-

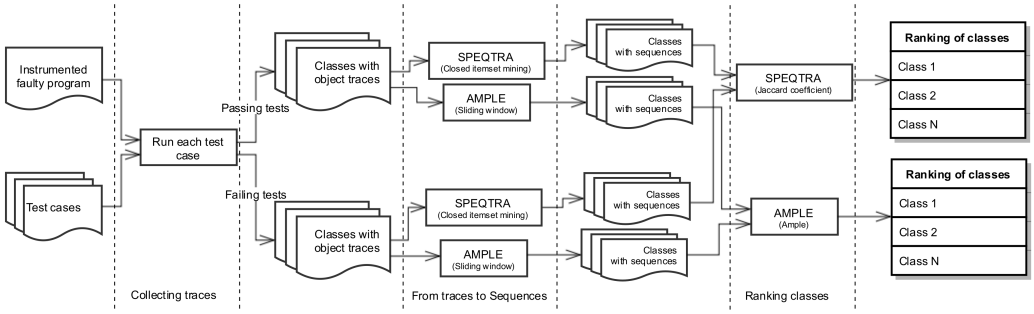


Figure 2.1: Overview of the two heuristics showing three steps (i) Collecting tracing, (ii) Collecting class sequences, and (iii) Ranking classes

cantly better than the one of AMPLE: we can immediately pinpoint the faulty class in 56% of all test runs; while on average 12% of the executed classes (5% of all classes) must be inspected to find the location of the fault. Moreover, for 70% of the faults, SPEQTRA has at most one false positive whereas for AMPLE this happens for 59% of the faults.

This chapter is structured as follows. We explain the two heuristics under investigation in Section 2.2. Next, we describe the way we set up our replication experiment, including the particular details about the NanoXML case in Section 2.3. Then the bulk of the chapter is contained within Section 2.4, where we show the results of the comparison including some anecdotal evidence from NanoXML. We list the related work, showing other research on dynamic analysis within a software evolution context in Section 2.5, followed by a discussion on the threats to validity in Section 2.6. Finally, Section 2.7 summarises our findings and lists the contributions.

2.2 HEURISTICS UNDER INVESTIGATION

In this section we give a detailed explanation of the two heuristics under investigation. First, we provide some background information regarding spectrum based fault localisation which is the basis for the two heuristics (Section 2.2.1). Then we contrast the two heuristics depicted in a Figure 2.1 showing where they are the same (i.e. collecting traces per object — Section 2.2.2) and where they differ (creating sequences of method calls — Section 2.2.3; the similarity coefficient used to rank the corresponding fault locations — Section 2.2.4).

2.2.1 Spectrum Based Fault Localisation

AMPLE and SPEQTRA both are instances of the class of spectrum based fault localisation heuristics [13]. Such heuristics discover statistical coincidences between system failures and the activity of the different parts of a system. All these heuristics create a so-called *program spectrum*, which is a matrix where each column corresponds to a program entity (e.g. statement, block, sequence of method calls) and rows represent a particular test run. For each test run the corresponding column for program entity is marked as 1 (executed) or 0 (not executed). Alongside the program spectrum, the heuristic also creates an *error vector* which is a column where a cell is marked as 1 if the test run failed or 0 if the test was a success. Next, the error vector is compared against all columns in the program spectrum using a particular *similarity coefficient*; the column which is most similar to the error vector is then the program entity which most likely contains the fault.

2.2.2 Collecting Traces

AMPLE and SPEQTRA use sequences of method calls as the program entities which are represented in each column of the fault spectrum matrix. Both heuristics group the outgoing method calls according to the following scheme. Let $O = \{o_1, o_2, ..., o_n\}$ be the set of object instances of the class C and $T = \{t_1, t_2, ..., t_n\}$ be the set of object traces of class C, where t_i represents the trace of outgoing method calls by the object o_i. By outgoing method calls we mean an object calling a method of another object. For instance if we have an object o_1 with a method `m1()` and hit method hosts a call to method `n1()` belonging to an object o_2, the collected outgoing method call for o_1 is `m1()`.

Two objects o_1 and o_2 of a class C may have following traces of method calls (Equations 2.1 and 2.2):

$$t_1 = \left\{ \begin{array}{c} m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3 \end{array} \right\} \tag{2.1}$$

$$t_2 = \{m_1, m_1, m_1\} \tag{2.2}$$

AMPLE and SPEQTRA group all such object traces for the corresponding class C which has trace set T (Equation (2.3)).

$$T = \left\{ \begin{array}{l} \{\, m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3\}, \\ \{\, m_1, m_1, m_1\} \end{array} \right\} \tag{2.3}$$

2.2.3 From Traces to Class Sequences

Traces of outgoing method calls can grow to millions of method calls per object [44]. To reduce these traces AMPLE and SPEQTRA each apply a different technique to arrive at what we call *Class Sequences* for the remainder of the chapter.

AMPLE — Sliding Window. AMPLE slides a window of fixed size over the trace to create a list of class sequences. From the previous example, if we fix the window size as 2 and slide it over the object traces in Equation (2.3) we obtain the set of class sequences in Equation (2.4)

$$C_A = \left\{ \begin{array}{l} \{\{m_1, m_1\}, \{m_1, m_2\}, \{m_2, m_2\}, \\ \{m_2, m_3\}, \{m_3, m_1\}\} \end{array} \right\} \tag{2.4}$$

SPEQTRA — Frequent Sequences. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [23]. Given the set of object traces T of class C, we define:

- X —*itemset*— a set of method calls.
- $\sigma(X)$ —support of X— the number of traces of T that contain this itemset X.
- $minsup$ —minimum support of X— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset X is frequent when $\sigma(X) \geq minsup$.
- *closed itemset* — a frequent itemset X is closed if there exists no proper superset X' whose support is same as the support of X (i-e. $\sigma(X') = \sigma(X)$).

From now on, we refer a closed itemset X as a frequent sequence or simply a sequence of method calls. Adopting closed itemset mining in the context of fault localisation, we fix $minsup$ to 1 because those classes which only create one object (and thus one trace) should be included in the program spectrum as well; this one call trace may be the one which triggers the fault. However, we tune the algorithm in another way. The mining algorithm also returns frequent sequences that comprise only one method call. Since we are looking for faults caused by complex object interactions where objects must adhere to a certain protocol, sequences should have at least a length of two.

From the previous example with input T (Equation (2.3)) and $minsup = 1$ the generated set of frequent sequences is:

$$C_F = \{\{m_1\}, \{m_1, m_3, m_2\}\} \tag{2.5}$$

It can be observed that the sequences such as $\{m_2\}$, $\{m_3\}$, $\{m_1, m_2\}$, $\{m_1, m_3\}$, $\{m_2, m_3\}$

are not included in final set (Equation (2.5)), since there exists a super sequence $\{m_1, m_3, m_2\}$ with equal support. As SPEQTRA removes all frequent sequences of length 1, thus the sequence set C_F (Equation (2.5)) is finally reduced into the set of Class Sequences C_S in (Equation (2.6)).

$$C_S = \{\{m_1, m_3, m_2\}\} \tag{2.6}$$

2.2.4 Ranking Classes

Both AMPLE and SPEQTRA assign a weight $W(X)$ to each class sequence in C_A and C_S(Equation (2.4) and 2.6 respectively). Note that X refers to a sequence of method calls, but there is a difference in that X is a chunk of fixed size in AMPLE whereas X is set of frequent method calls in SPEQTRA.

AMPLE — AMPLE has defined its own weighting scheme based on a configuration of a single failing test and several passing tests. Sequences in AMPLE are assigned a weight between 0 and 1 using equation (2.7) [22].

$$W(X) = \begin{cases} \frac{k(X)}{n} & \text{if } X \text{ not in failing test} \\ 1 - \frac{k(X)}{n} & \text{if } X \text{ in failing test} \end{cases} \tag{2.7}$$

Where n is the number of passing tests and $k(X)$ is the number of passing tests that include the sequence X.

SPEQTRA — We tested several weighting schemes to rank the classes. Ultimately, in SPEQTRA, we opted for the Jaccard similarity coefficient (Equation (2.8)) adopted from Chen et al. [45]:

$$W(X) = \frac{a_{11}(X)}{a_{11}(X) + a_{01}(X) + a_{10}(X)} \tag{2.8}$$

Where:

- $a_{11}(X)$ = Number of failing tests in which $sequenceX$ is found.
- $a_{10}(X)$ = Number of passing tests in which $sequenceX$ is found.
- $a_{01}(X)$ = Number of failing tests in which $sequenceX$ is not found.

Weight per class. Both AMPLE and SPEQTRA take the average of all weights for all sequences of a class and assign this weight to class C, as defined in Equation (2.9):

$$W(C) = \frac{1}{n} \sum_{i=1}^{n} W(X_i) \tag{2.9}$$

where n is the number of sequences in the class and $W(X_i)$ is weight of a sequence as

Table 2.1: NanoXML version details

Version	Number of classes	LOC	Number of faults	Number of tests
1	16	4334	7	214
2	19	5806	7	214
3	21	7185	10	216
5	23	7646	8	216

given by Equation (2.7) (AMPLE) or Equation (2.8) (SPEQTRA).

Finally, both heuristics rank all classes using their weights $W(C)$, where the one with the highest weight is the most likely location of the fault.

2.3 EXPERIMENTAL SETUP

This chapter is set-up as a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. In what follows, we describe the way we set up our replication experiment, including the particular details about the NanoXML case (Section 2.3.1). Next, we provide the necessary practical details about the mechanics of the experiment so that other researcher can replicate our findings (Section 2.3.2)

2.3.1 Replication Case --- NanoXML

The original paper proposing the AMPLE heuristic demonstrated its effectiveness on a small but representative project named NanoXML [22]. NanoXML is a non-validating XML parser written in Java. Its source code and documentation are available in the Software-artifact Infrastructure Repository[1] [46].

NanoXML has five development versions (V1 ... V5) where the number of classes span from 16 to 23 (Table 3.2). With the exception of version V4, all others have documented faults that can be activated and exposed by the test suite. These versions (V1,V2,V3,V5) —the ones we use for the experiment— have 32 faults (cumulatively). Each version is shipped along with tests and test drivers. A test driver is a class that sets up multiple tests by feeding them with the required input (e.g. read a XML file). The goal of these

[1] http://sir.unl.edu/portal/index.php

test drivers is to trigger one and only one feature of the project.

We collect the traces of all faults by injecting one fault at a time and subsequently running the test suite. Then for each test, we record the outgoing methods calls of the created objects. It is important to note that, for each test (passing or failing), a separate trace is maintained for each object. All the outgoing method calls of an object appear in their own trace. Thus, two objects o_1 and o_2 of the same class C have independent traces of method calls.

In the replication experiment we activate one fault at a time. Having 32 faults leads to 32 distinct variants of NanoXML. Among these variants we picked only the ones that generate at least one failing and one passing test for each test driver. Just like in the original AMPLE experiment, this ensures that failing and passing tests are all related to the same functionality. For each test driver, we group one failing test with all passing tests, the set-up that is needed to reflect the set-up of AMPLE experiment. We repeat this process for each failing test associated to that test driver. At the end of this process, we end up with 18 variants of NanoXML and 347 combinations of failing and passing tests used for our experiments.

Note that this set-up is not exactly the same as the one reported in the AMPLE paper because the version of NanoXML we downloaded from the Software-artifact Infrastructure Repository has been changed. In the latest version, one fault is removed from V5 with a note "since it is overly expressive it may not be representative of a *pseudo-real* fault". As a consequence, the fault matrix also differs from the previous version. This is an inherent risk with replication experiments and partially explains why we do not obtain the same results reported in the AMPLE experiment [22].

2.3.2 Replication Details

AMPLE replication. When preparing for the replication experiment, we downloaded the original binary of the AMPLE implementation. Due to hardware constraints, we were unable to run this binary. Consequently, we implemented our own version of the algorithm as reported in the original AMPLE paper [22]. We used the optimal settings for the parameters of the heuristic, in particular we adopted a sliding window size of 8.

AspectJ. The object traces are collected by introducing logger functionality into the NanoXML code via AspectJ[2]. More specifically, we use a method call join point with a pointcut to pick out every call site. Each time a method call occurs, the aspect extracts the caller object and adds a method entry to the object's trace. The aspect is robust for different threads that may be running within the java project, although this was irrelevant

[2]AspectJ `http://eclipse.org/aspectj/`

for the NanoXML case. All object traces belonging to same class appear together in a HashMap maintained for each executed class.

Static Methods are Ignored. SPEQTRA, like AMPLE, also collects traces of method calls invoked by objects of a class. Calls to static methods are not captured and do not appear in the trace, hence cannot be identified as the location of the fault. For the particular replication of the NanoXML experiment this did not cause any problems however this limitation must be taken into account for future replication.

Single failing test. For this experiment, we inject one fault into the program which causes one or more tests to fail and several of them to pass. All these tests are executed with same test driver. In principle, SPEQTRA is able to rank fault locations using all these failing and related passing tests. However, since AMPLE is designed to work with only one failing test we replicated the set-up to include the trace of a single failing test and one or more passing tests.

Closed Itemset Mining. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [23]. In particular, we used the implementation provided by the library SPMF[3].

Search Length. To compare the results of the two heuristics we use the so-called *search length* as defined in the AMPLE experiment [22]. The search length counts how many classes are placed atop of the faulty class in the ranking produced by the heuristic. In that sense, it represents how many classes the developer has to examine before finding the class containing the fault. The search length is zero whenever the faulty class is placed as the first item in the ranking.

2.4 RESULTS AND DISCUSSION

To compare our results with AMPLE, we replicated AMPLE with a sliding window size 8, which is the value for which AMPLE achieved the best performance. Following the experimental set-up explained in Section 2.3, we obtained rankings for all the 347 combinations both with our implementation of AMPLE and SPEQTRA. Below we discuss the results of both.

1. Our replication experiment confirmed the results reported in the original AMPLE paper. There were some minor changes in the results, but these can be attributed to the differences in the NanoXML version used in our experiment, in particular in the tests accompanying the project.

[3] SPMF `http://www.philippe-fournier-viger.com/spmf/`

Table 2.2: Average Search Length in SPEQTRA and AMPLE

AMPLE	SPEQTRA
2.07	1.20

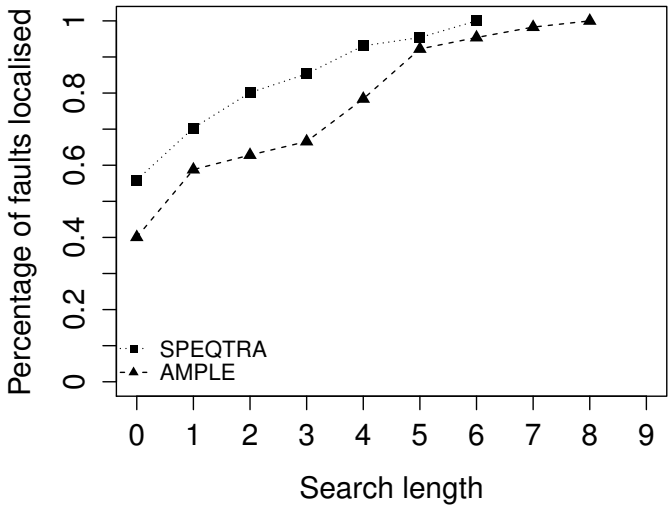


Figure 2.2: Search Length in SPEQTRA and AMPLE.

2. The average search length of all rankings in 347 test runs with both heuristics is reported in Table 2.2. Here SPEQTRA has less average search length than sliding window approach.
3. In 173 test runs out of 347, SPEQTRA outperforms AMPLE and has search length less than AMPLE. In 140 test runs both SPEQTRA and AMPLE have same search length, whereas in only 34 cases SPEQTRA has search length greater than AMPLE.
4. The plot of the cumulative of search length distribution in 347 test runs with both heuristics is given in Figure 2.2. With SPEQTRA, the search length of 0 covers 56% of faults, whereas with AMPLE it is only 40% of faults. Furthermore, the worst case search length with SPEQTRA is 6 whereas with AMPLE it is 8.

Listing 2.1: Code snippet with a fault in XMLElement class

```
1  public Enumeration enumerateAttributeNames() {
2   Vector result = new Vector();
3   Enumeration _enum = this.attributes.elements();
4   while (_enum.hasMoreElements()) {
5    XMLAttribute attr = (XMLAttribute) _enum.nextElement();
6    // The call should be to attr.getName()
7    result.addElement(attr.getFullName());
8   }
9   return result.elements();
10 }
```

2.4.1 Anecdotal Evidence

From the experiment we collected some anecdotal evidence highlighting the main differences between the two approaches. Specifically the two differences, namely (i) the effect of the sliding window and (ii) the impact of repetitive method calls (e.g. contained in loops). In Listing 2.1, we see a piece of source code showing a fault in XMLElement class at line 7 which was exercised by several unit tests. This resulted in three XMLElement object traces generated by the failing test as shown in Listings 2.2, 2.3 and 2.4. Note that for brevity, method parameters are not shown.

From these object traces in failing test, SPEQTRA generated three sequences of method calls for class XMLElement . For brevity, we list the numbers in the sequences instead of method calls. These numbers in the sequences represents the line numbers in Listing 2.2, unless otherwise mentioned. The line number for the method is the very first entry of the method in the trace. The first sequence was s1 = {1, 4, 21, 24, 5, 15, 7}, the methods indicated by numbers 5 and 7 appear in Listing 2.3 at lines 5 and 7. The second sequence was s2 ={1, 4, 13, 21, 24, 15} and the third sequence was s3 = {1, 4, 21, 24, 15}. These sequences capture frequent method calls occurring in the traces and hence represent a good abstraction of the traces. The three sequences have length 7, 6 and 5 respectively and none of the sequence has repetitive method calls.

On the other hand, AMPLE generated 37 sequences for the class, each with repetitive method calls. If we slide a window of size 8 over the trace in Listing 2.2, the first sequence s1 = {1, 2, 3, 4 ,5, 6, 7, 8} contains 5 repetitions for method XMLElement.findAttribute() and three repetitions of method XMLAttribute.getFullName(). Likewise the second sequence s2 = {2, 3, 4 ,5, 6, 7, 8, 9} contains four times method XMLElement.findAttribute() and 4 times method XMLAttribute.getFullName(). As a consequence, the AMPLE heuristic fails to locate the fault and it ranked the faulty class on position 6 whereas SPEQTRA could pinpoint it exactly.

Listing 2.2: Failing trace of XMLElement object 1

```
1  XMLElement.findAttribute(String)
2  XMLElement.findAttribute(String)
3  XMLElement.findAttribute(String)
4  XMLAttribute.getFullName()
5  XMLElement.findAttribute(String)
6  XMLAttribute.getFullName()
7  XMLElement.findAttribute(String)
8  XMLAttribute.getFullName()
9  XMLAttribute.getFullName()
10 XMLElement.findAttribute(String)
11 XMLAttribute.getFullName()
12 XMLAttribute.getFullName()
13 XMLElement.getName()
14 XMLElement.getName()
15 XMLAttribute.getName()
16 XMLAttribute.getName()
17 XMLAttribute.getName()
18 XMLAttribute.getName()
19 XMLAttribute.getName()
20 XMLAttribute.getName()
21 XMLElement.getAttribute()
22 XMLElement.findAttribute(String)
23 XMLAttribute.getFullName()
24 XMLAttribute.getValue()
25 XMLElement.getAttribute()
26 XMLElement.findAttribute(String)
27 XMLAttribute.getFullName()
28 XMLAttribute.getFullName()
29 XMLAttribute.getValue()
30 XMLElement.getAttribute()
31 XMLElement.findAttribute(String)
32 XMLAttribute.getFullName()
33 XMLAttribute.getFullName()
34 XMLAttribute.getFullName()
35 XMLAttribute.getValue()
```

There are however 34 situations where AMPLE was more accurate than SPEQTRA. We use one fault injected in the class NonValidator to explain this difference. The failing test and several of the passing tests generated the same trace for the only object of NonValidator. In this case, SPEQTRA generated one sequence for the failing test and two sequences for the passing tests, one of which also appeared in the failing test. As a consequence, SPEQTRA ranked this class on position 4 whereas AMPLE could pinpoint it exactly. This can be explained as, the Jaccard similarity coefficient (or any other coefficient used in spectrum based fault localisation) assigns more weight to a sequence when it is present more in failing tests and less in passing tests. Hence the sequence only present in passing tests gets weight 0; the value 0 for numerator $a_{11}(X)$ in equation 2.8 evaluates the whole equation to 0. The other sequence presented in failing test gets a lower weight due to its

presence in several passing tests; the higher value of denominator $a_{10}(X)$ in equation 2.8 decreases the value. Consequently, the weight of the class, which is average weight of the two sequences, is also less.

Listing 2.3: Failing trace of XMLElement object 2

```
1  XMLElement.findAttribute(String)
2  XMLElement.findAttribute(String)
3  XMLElement.findAttribute(String)
4  XMLAttribute.getFullName()
5  XMLElement.findAttribute(String, String)
6  XMLAttribute.getName()
7  XMLAttribute.getNamespace()
8  XMLAttribute.getName()
9  XMLAttribute.getName()
10 XMLAttribute.getName()
11 XMLAttribute.getName()
12 XMLElement.getAttribute( )
13 XMLElement.findAttribute(String)
14 XMLAttribute.getFullName()
15 XMLAttribute.getValue()
16 XMLElement.getAttribute( )
17 XMLElement.findAttribute(String)
18 XMLAttribute.getFullName()
19 XMLAttribute.getValue()
```

Listing 2.4: Failing trace of XMLElement object 3

```
1  XMLElement.getName()
```

2.4.2 Discussion

Based on this replication experiment, we conclude that SPEQTRA is significantly better than AMPLE since:

1. The average ranking in SPEQTRA is lower than AMPLE. This average suggests that a developer has to search through, on average, 12% of 10.25 average executed classes or 5% of all 23 classes. This is significantly better than AMPLE, where 20% of executed classes or 9% of all classes need to be searched.
2. A faulty class is placed first in the ranking (search length 0) for 56% of faults by SPEQTRA whereas for AMPLE it happens only for 40% of the faults.
3. For 70% of faults, there is atmost one false positive (search length 1) with SPEQTRA whereas for AMPLE this happens for 59% of the faults.

2.5 RELATED WORK

In this section, we present related work on spectrum based fault localisation thus immediately relevant for this replication experiment. Moreover, we also give references to related work on dynamic analysis for program comprehension as this provides the broader context for our research.

2.5.1 Spectrum Based Fault Localisation

Spectrum based fault localisation is an automated fault diagnosis technique based on differences in program spectra of a program between passing and failing tests [13]. Spectrum based fault localisation techniques have been applied in many domains such as localising the fault and ranking program statements [12], blocks [13], failure related components [45] and —last but not least— classes [22].

Jones and Harrold used statement-hit spectra to rank statements of C programs according to their likelihood to be at fault [12]. To visualize the ranking, they implemented a tool — Tarantula — able to mark statements with colors that span from red (statement likely at fault) to green (statement unlikely at fault).

Abreu et al. used blocks-hit spectra to rank the blocks in order of their likelihood to be at fault [13]. Here the block is defined as C language statement where compound statement (statements inside curly brackets) counts as a single statement. They compared the performance of different similarity coefficients and their impact on diagnostic accuracy of spectrum based fault localisation technique.

Chen et al. implemented the tool Pinpoint for tracing client requests in Internet service environments. Pinpoint records the components involved in the service and whether or not the request is satisfied [45]. The tool correlates the request failures to the components that most likely caused the failure.

All previous papers detect the fault at different levels of granularity(e.g., statements, blocks). However, none of them uses spectrum based fault localisation to identify faults due to method call sequences. In the literature, the only two techniques able to localise faults related to method call sequence are AMPLE and MCA-E. AMPLE is a tool, created by Dallmeier et al. that traces method calls invoked by objects and collects call sequences of the corresponding classes [22]. The outcome of the tool is a list of classes ranked according to their likelihood to be at fault. MCA-E is a technique proposed by Tu et al. in order to improve the regular spectrum based fault localisation techniques adopted in AMPLE [47]. Its outcome is a list of statements ranked according to their likelihood to be at fault. In the first step, it computes the likelihood of classes to be at fault (suspiciousness) by taking into account the difference of their method call sequences between passing and failing

tests. In the second step, the suspiciousness of classes is used together with their statements to generate ranking of statements. One of the limitations of AMPLE and MCA-E approaches is the adoption of a window of finite size that slides over the execution traces. Such sliding a window is not efficient for computing the sequences since method calls may stem from loops or may repeat in a trace resulting into sequences with repetitive method calls which add overhead with little extra information. Furthermore, the number of sequences linearly increases as the size of the trace increases. With window size w and n number of method calls in a trace, n^w sequences are possible. In this chapter, we address the shortcomings related to the sliding window by mining the frequent method call sequences. The sequence mining alleviates the arbitrary upper limit on the length of the call sequence. It also optimises the computational power required to obtain the ranking of classes as the mining algorithm limits the frequent sequences to closed ones: and also the SPEQTRA removes one-length sequences, the number of SPEQTRA sequences is far less than sliding a window over the trace.

It also optimises the computational power required to obtain the ranking of classes as it generates far less sequences than sliding a window over the trace.

2.5.2 Program Comprehension

Discovering program invariants and specifications such as legal method call sequences are common goals of research in program comprehension [41, 48, 49]. Such specifications, achieved by means of dynamic analysis, are used for purposes including documentation, learning the API's etc.

Ernst et al. implemented the tool Daikon to dynamically detect program invariants [48]. An invariant is defined as a property that is true at a particular program point or points. Daikon runs an instrumented program over a series of test runs and records program properties. At the invariant detection stage it starts with a list of hypothetical invariants comparing them across all the traced properties of the program for all test runs. It immediately discards the hypothetical invariant the moment it does not hold for a test run. Finally, all the invariants that are validated across all test runs are reported. Daikon can also be used to detect invariant violations in failing tests and as such may be used in a similar set-up as what we report here.

Gabel and Su implemented OCD, a tool which traces method calls and, using a predefined template as a model for specification inference, learns and enforces temporal specifications over method call sequence [49]. The algorithm suffers from two limitations: (1) the template limits the sequence to comprise only two method calls and (2) the sequences inferred from a limited window size. The efficiency of the algorithm critically depends on the window size. Experimenting with Eclipse and Ant, the tool detected a few anoma-

lies as violations of inferred sequences, though the anomalies did not result in program crashes.

Pradel and Gross proposed a dynamic analysis technique to infer specifications of correct method call sequences [41]. The technique focuses on object collaboration, namely objects and method calls used together in the execution of a single method. By running a software program, the technique traces method calls, computes object collaborations and identifies patterns among these collaborations. From these patterns, the technique infers the legal method call sequence in the form of finite state machines.

To certain extent, our research on SPEQTRA is complementary to the previous ones. We use method call sequences of a class from passing tests, which can be assumed as usage patterns of the program. On the other hand, the sequences in failing tests can be considered as deviant behaviour.

2.6 THREATS TO VALIDITY

Following the template for case studies in [50], we discuss the threats to validity that can affect our results.

Threats to **external validity** correspond to the generalizability of our experimental results. Our study is limited to the object oriented system NanoXML. Although NanoXML is small project, it represents a good testbed since it provides documented tests and faults for replicating our study. Moreover, by using the same case study of Dallmeier et al. [22], we were able to verify the impact of removing the sliding window and adopting different similarity coefficients for mining faults in stack traces. Nevertheless, it is desirable to replicate our findings using other projects.

Threats to **internal validity** concern confounding factors that can influence the obtained results. Our approach leverages on the fault's "ability" of changing the stack trace generated by software execution. In that sense, we localise faults by pointing out the class that has different method call sequences (in passing and failing tests) assuming that such deviation is due to the fault. This assumption is a key-element in spectrum based fault localisation techniques based on method call sequences [22, 47] and our results confirm its general validity. On the other hand, there are cases where it does not apply. In NanoXML there is (only) one faulty class that cannot be localised —with our approach— since the fault is caused by a variable accessed without any method call.

Threats to **construct validity** focus on how accurately the observations describe the phenomena of interest. Our experiment relies on the correct identification of fault responsible for test failure. From this point of view, we do not have threats to construct validity since we inject one fault at the time. When the fault is injected, otherwise the test passes.

Threats to **reliability validity** correspond to the degree to which the same data would lead to the same results when repeated. We describe all steps of our technique and provide references on any tool or library involved in the analysis. The case study we use is publicly available in the Software-artifact Infrastructure Repository[4], a repository created for supporting rigorous controlled experimentation with program analysis and software testing techniques [46].

2.7 CONCLUSION

In this chapter, we presented a novel spectrum based fault localisation heuristic (named SPEQTRA) which used closed itemset mining to identify the characteristic method call sequences and the Jaccard similarity coefficient to rank the classes according to the likelihood of containing the fault. We compare our fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (NanoXML) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE. In particular, SPEQTRA can immediately pinpoint the faulty class in 56% of all test runs (against 40% for AMPLE); while on average 12% of the executed classes must be inspected to find the location of the fault (against 20% for AMPLE). From anecdotal evidence, we deduce that the main reason why SPEQTRA performs better than AMPLE is due to closed itemset mining: this filters out repetitive method calls (e.g. contained in loops) and avoids the arbitrary upper limit imposed by the sliding window. Nevertheless, for a few faults AMPLE provides a better ranking than SPEQTRA, caused by call sequences appearing in both failing and many passing tests which reduced the weight of sequences.

Over the course of this research, we have made the following contributions:

- *Replication Experiment.* We conducted a replication of the AMPLE experiment performed by Dallmeier et al. [22]. We used the same data (the NanoXML case provided in the Software-artifact Infrastructure Repository [46]) and confirmed the numbers provided in the original report.
- *Alternative Heuristic.* We proposed an alternative spectrum based fault localisation heuristic (named SPEQTRA). We compared it against the results from AMPLE. We demonstrate that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.
- *Anecdotal Evidence.* We collected some anecdotal evidence from the NanoXML case interpreting the main differences between the two heuristics.

Fault localisation heuristics are particularly relevant in modern software engineering owing to the increasing popularity of continuous integration. Continuous integration

[4] http://sir.unl.edu/portal/index.php

states that software engineers should merge their working copies with the main branch several times a day using a suite of automated tests to verify the correctness of the build. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In that sense our work shows that while the state of the art is rapidly advancing, it is worthwhile to make improvements on research from a decade ago.

2.8 ACKNOWLEDGMENTS

This work is sponsored by (i) the Higher Education Comission of Pakistan under a project titled “Strengthening of Univeristy of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled “Change-centric Quality Assurance (CHAQ)” with number 120028.

Chapter 3

Fine-tuning Spectrum Based Fault Localisation with Frequent Method Item Sets

Fine-tuning Spectrum Based Fault Localisation with Frequent Method Item Sets
Gulsher Laghari, Alessandro Murgia, and Serge Demeyer
In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 274–285. Singapore, Singapore. September, 2016.
DOI: https://doi.org/10.1145/2970276.2970308.

This chapter was originally published in the *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*.

ABSTRACT

Continuous integration is a best practice adopted in modern software development teams to identify potential faults immediately upon project build. Once a fault is detected it must be repaired immediately, hence continuous integration provides an ideal testbed for experimenting with the state of the art in fault localisation. In this chapter we propose a variant of what is known as spectrum based fault localisation, which leverages patterns of method calls by means of frequent itemset mining. We compare our variant (we refer to it as patterned spectrum analysis*) against the state of the art and demonstrate on 351 real faults drawn from five representative open source java projects that* patterned spectrum analysis *is more effective in localising the fault.*

3.1 INTRODUCTION

Continuous integration is an important and essential phase in a modern release engineering pipeline [5]. The quintessential principle of continuous integration declares that software engineers should frequently merge their code with the project's codebase [34, 51]. This practice is helpful to ensure that the codebase remains stable and developers can continue further development, essentially reducing the risk of arriving in *integration hell* [52]. Indeed, during each integration step, a continuous integration server builds the entire project, using a fully automated process involving compilation, unit tests, integration tests, code analysis, security checks, ... When one of these steps fails, the build is said to be *broken*; development can then only proceed when the fault is repaired [35, 36].

The safety net on automated tests, encourages software engineers to write lots of tests — several reports indicate that there is more test code than application code [38, 39, 53]. Moreover, executing all these tests easily takes several hours [40]. Hence, it should come as no surprise that developers defer the full test to the continuous integration server instead of running them in the IDE before launching the build [54]. Occasionally, changes in the code introduce regression faults, causing some of the previously passing test cases to fail [55]. Repairing a regression fault seems easy: the most recent commits should contain the root cause. In reality it is seldom that easy [36]. There is always the possibility of lurking faults, i.e. faults in a piece of code which are revealed via changes in other parts of the code [56]. For truly complex systems with multiple branches and staged testing, faults will reveal themselves later in the life-cycle [57, 58].

Luckily, the state-of-the-art in software testing research provides a potential answer via *spectrum based fault localisation*. These heuristics compare execution traces of failing and passing test runs to produce a ranked list of program elements likely to be at fault. In this chapter, we present a variant which leverages patterns of method calls by means of frequent itemset mining. As such, the heuristic is optimised for localising faults revealed by integration tests, hence ideally suited for serving in a continuous integration context.

In this chapter, we make the following contributions.

1. We propose a variant of spectrum based fault localisation (referred to as patterned spectrum analysis in the remainder of this chapter) which leverages patterns of method calls by means of frequent itemset mining.
2. We compare patterned spectrum analysis against the current state-of-the-art (referred to as raw spectrum analysis in the remainder of this chapter) using the Defects4J dataset [59].
3. The comparison is inspired by a realistic fault localisation scenario in the context of continuous integration, drawn from a series of discussions with practitioners.

Table 3.1: An Example Test Coverage Matrix and Hit-Spectrum

Element undet test	Failing tests	Passing tests	e_f	e_p	n_f	n_p
	$t_1 \ \ldots \ t_m$	$t_{m+1} \ \ldots \ t_n$				
$unit_i$	$X_{i,1} \ldots X_{i,m}$	$X_{i,m+1} \ldots X_{i,n}$	$\sum_{j=1}^{m} X_{i,j}$	$\sum_{j=m+1}^{n} X_{i,j}$	$m - e_f$	$(n-m) - e_p$

t_i denotes i_{th} test case $X_{j,j}$ takes the binary value 0 or 1

Table 3.2: Popular Fault Locators

Faul Locator	Definition
Tarantula [12]	$\frac{\frac{e_f}{e_f + n_f}}{\frac{e_f}{e_f + n_f} + \frac{ep}{e_p + n_p}}$
Ochiai [13]	$\frac{e_f}{\sqrt{(e_f + n_f)(e_f + e_p)}}$
T* [60]	$\left(\frac{\frac{e_f}{e_f + n_f}}{\frac{e_f}{e_f + n_f} + \frac{ep}{e_p + n_p}} \right) \times max\left(\frac{e_f}{e_f + n_f}, \frac{e_p}{e_p + n_p} \right)$
Naish2 [17]	$e_f \ - \ \frac{e_p}{e_p + n_p + 1}$

The remainder of this chapter is organised as follows. Section 3.2 lists the current state-of-the-art. Section 3.3 presents a motivating example, followed by Section 3.4 explaining the inner details of our variant. Section 3.5 describes the case study set-up, which naturally leads to Section 3.6 reporting the results of the case study. After a discussion of potential improvements in Section 3.7, and the threats to validity in Section 3.8, we come to a conclusion in Section 3.9.

3.2 STATE OF THE ART

This section provides an overview of the current state-of-the-art in spectrum based fault localisation. In particular, we sketch the two dimensions for the variants that have been investigated: either the granularity (statement — block — method — class) or the fault locator function (Tarantula, Ochiai, T*, and Naish2). We also explain what is commonly used when evaluating the heuristics: the evaluation metric (*Wasted Effort*) and the available benchmarks and datasets. Finally, we list some common applications of fault

localisation, which heavily influences the way people assess the effectiveness in the past research.

Automated Fault Localisation. To help developers quickly locate the faults, there exist two broad categories of automated fault localisation heuristics: (1) information retrieval based fault localisation [7, 8, 9, 10], and (2) spectrum based fault localisation [11, 12, 13, 60, 61]. Both of these categories produce a ranked list of program elements, indicating the likelihood of a program element causing the fault. While the former uses bug reports and source code files for analysis, the later uses program traces generated by executing failing and passing test cases. Since spectrum based fault localisation heuritsics only require traces from test runs— readily available after running the regression test suite— these heuritics are ideally suited for locating regression faults in a continuous integration context.

Spectrum based fault localisation is quite an effective heuristic as reported in several papers [12, 13, 60, 61]. Sometimes other names are used, namely coverage based fault localisation [62] and statistical debugging [63]. To understand how spectrum based fault localisation heuristics work, there are three crucial elements to consider: (1) the test coverage matrix; (2) the hit-spectrum; and the (3) fault locator. We explain each of them below.

1. All spectrum based fault localisation heuristics collect coverage information of the elements under test in a *test coverage matrix*. This is a matrix, where the rows correspond to elements under test and the columns represent the test cases. Each cell in the matrix marks whether a given element under test is executed (i.e. covered) by the test case (marked as 1) or not (marked as 0).

2. Next, this test coverage matrix is transformed into the *hit-spectrum* (sometimes also called *coverage spectrum*) of a program. The hit-spectrum of an element under test is tuple of four values (e_f, e_p, n_f, n_p). Where e_f and e_p are the numbers of failing and passing test cases that execute the element under test and n_f and n_p are the numbers of failing and passing test cases that do *not* execute the element under test. Table 3.1 shows an example test coverage matrix and spectrum.

3. Finally, the heuristic assigns a suspiciousness to each element under test by means of a *fault locator*. This suspiciousness indicates the likelihood of the unit to be at fault. The underlying intuition is that an element under test executed more in failing tests and less in passing tests gets a higher suspiciousness and appears at top position in the ranking. Sorting the elements under test according to their suspiciousness in descending order produces the ranking. Many (if not all) variants of spectrum based fault localisation create a new fault locator; Table 3.2 gives an overview of the most popular ones.

Granularity. Other variants of spectrum based fault localisation concern the choice of the elements under test. Indeed, spectrum based fault localisation has been applied at different levels of granularity, including *statements* [11, 12, 64, 65, 66], *blocks* [13, 16, 67, 68, 69], *methods* [60, 61, 70], and *classes* [22, 30]. The seminal work on spectrum based fault localisation started off with statement level granularity [11]. As a result, most of the early research focussed at statement level, sometimes extended to basic blocks of statements. The effectiveness at the method level has been investigated in only a few cases and then even as part of a large-scale comparison involving several levels of granularity [60, 61, 70].

Today, the space of known spectrum based fault localisation heuristics is classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T, and Naish2). In this chapter, we explore the hit-spectrum as a third dimension. We expand the four tuple (e_f, e_p, n_f, n_p) so that e_f and e_p incorporate patterns of method calls we extracted by means of frequent itemset mining.*
In the remainder of this chapter we refer to the current state of the art as raw spectrum analysis, while our variant will be denoted with patterned spectrum analysis.

Evaluation metric: wasted effort. Fault localisation heuristics produce a ranking of elements under test; in the ideal case the faulty unit appears on top of the list. Several ways to evaluate such rankings have been used in the past, including relative measures in relation to project size, such as the percentage of units that need or need not be inspected to pinpoint the fault [60]. Despite providing a good summary of the accuracy of a heuristic, absolute measures are currently deemed better for comparison purposes [60, 61, 71]. Today, the *wasted effort* metric is commonly adopted [60, 61, 70]. Consequently, we will rely on the *wasted effort* when comparing raw spectrum analysis against patterned spectrum analysis. (The exact formula for wasted effort is provided in Section 3.5 — Equation (3.8)).

Dataset. The early evaluations on the effectiveness of raw spectrum analysis heuristics were done by means of small C programs, taken from the Siemens set and Space [72]. Despite having industrial origins, the faults used in the experiments were manually seeded by the authors [11, 73]. The next attempt at a common dataset for empirical evaluation of software testing and debugging is the Software-Artifact Infrastructure Repository (SIR) [46]. Unfortunately, most of the faults in this dataset are manually seeded as well. Consequently, Dallmeier and Zimmermann created the iBugs dataset containing real faults drawn from open source Java projects [74]. iBugs contains 223 faults all accompanied with at least one failing test case to reproduce the fault. The last improvement on

fault datasets is known as Defects4J [59]. Defects4J has a few advantages over iBugs: all the faults in Defects4J are isolated— the changes in V_{fix} for corresponding V_{bug} purely represent the bug fix. Unrelated changes —such as adding features or refactorings— are isolated. Defects4J also provides a comprehensive test execution framework, which abstracts away the underlying build system and provides a uniform interface to common build tasks — compilation, test runs, etc.... To the best of our knowledge, the Defects4J has not yet been used for evaluating raw spectrum analysis. Hence, we will adopt the Defects4J dataset for our comparison.

The current state of the art relies on wasted effort to evaluate fault localisation heuristics mainly via the SIR and iBugs datasets. When comparing raw spectrum analysis against patterned spectrum analysis, we rely on wasted effort as well, yet adopt the more recent Defects4J dataset.

Applications. In the initial research papers, the main perspective for spectrum based fault localisation was to assist an individual programmer during debugging [11, 12, 13, 16, 64, 66, 67, 68, 69]. The typical scenario was a debugging window showing not only the stack trace but also a ranked list of potential locations for the fault, hoping that the root cause of the fault appears at the top of the list. This explains why the accuracy of these heuristics was mainly evaluated in terms of percentage of code that needs to be inspected. Recently, another application emerged: automated fault repair [55, 75, 76]. The latter techniques repair a fault by modifying potentially faulty program elements in brute-force manner until a valid patch —i.e. one that makes the tests pass— is found. The first step in automated repair is fault localisation, which in turn resulted in another evaluation perspective, namely whether it increases the effectiveness of automated fault repair [77].

The two commonly used applications for fault localisation are debugging and automated fault repair. Up until now, continuous integration has never been considered. We will present the implications of broken builds within continuous integration in Section 3.3.

3.3 MOTIVATING SCENARIO

Since we propose continuous integration as a testbed for validating patterned spectrum analysis, it is necessary to be precise about what exactly constitutes a continuous integration tool and what kind of requirements it imposes on a fault localisation heuris-

tic. As commonly accepted in requirements engineering, we specify the context and its requirements by means of a *scenario*. The driving force underlying the scenario is the observation that if a build is broken, it should be repaired immediately hence the root cause should be identified as quickly as possible.

Note that at a first glance this scenario may seem naive. Nevertheless, it is based on a series of discussions with software engineers working with the agile development process SCRUM and who rely on a continuous integration server to deploy their software on a daily basis. The discussions were held during meetings of the steering group of the Cha-Q project (`http://soft.vub.ac.be/chaq/`), where we confronted practitioners with the scenario below and asked for their input on what a good fault localisation method should achieve. Therefore, we can assure the reader that the scenario represents a real problem felt within today's software teams.

Prelude: GeoPulse GeoPulse[1] is an app which locates the nearest location of an external heart defibrillator so that in case of an emergency one can quickly help an individual suffering from a cardiac arrest. The software runs mainly as a web-service (where the database of all known defibrillators is maintained), yet is accessed by several versions of the app running on a wide range of devices (smart phones, tablets and even a mini-version for smart watches).

Software Team. There is a 12 person team responsible for the development of the GeoPulse app; 10 work from the main office in Brussels while 2 work from a remote site in Budapest. The team adopts a SCRUM process and uses continuous integration to ensure that everything runs smoothly. It's a staged build process, where the build server performs the following steps: (1) compilation; (2) unit tests; (3) static code analysis; (4) integration tests; (5) platform tests; (6) performance tests; (7) security tests. Steps (1) — (3) are the level 1 tests and fixing problems there is the responsibility of the individual team members; steps (4) — (7) are the level 2 defence and the responsibility of the team.

Scene 1: Unit Testing. Angela just resolved a long standing issue with the smart-watch version of the app and drastically reduced the response time when communicating with the smart-phone over bluetooth. She marks the issue-report as closed, puts the issue-ID in the commit message and sends everything off to the continuous integration server. A few seconds later, the lava-lamp in her cubicle glows orange, notifying a broken build. Angela quickly inspects the build-log and finds that one unit-test fails. Luckily, the guidelines for unit tests are strictly followed within the GeoPulse team (unit-tests run fast, have few dependencies on other modules and come with good diagnosing messages). Angela can quickly pinpoint the root cause as a missing initialisation routine in one of the subclasses she created. She adds the initialiser, commits again and this time

[1]The name and description of the app is fictitious.

the build server finds no problems and commits her work to the main branch for further testing during the nightly build. The lava-lamp turns green again and Angela goes to fetch a coffee before starting her next work item.

Purpose. This scene illustrates the importance of the Level 1 tests and the role of unit tests in there. Ideally, running the whole suite of unit tests takes just a few seconds and if one of the unit tests fails, it is almost straightforward to locate the fault. Moreover, it is also clear who should fix the fault, as it is the last person who made a commit on the branch. Thus, fault localisation in the context of unit tests *sensu stricto* is pointless: the fault is located within the unit by definition and the diagnosing messages combined with the recent changes is usually sufficient to repair efficiently.

Scene 2: Integration Testing. Bob arrives in his office in the morning and sees that the lava-lamp is purple, signifying that the nightly build broke. He quickly inspects the server logs and sees that the team resolved 9 issues yesterday, resulting in 8 separate branches merged into the main trunk. There are three seemingly unrelated integration tests which fail, thus Bob has no clue on the root cause of the failure. During the stand-up meeting the team discusses the status of the build, and then suspends all work to fix the broken build. Team members form pairs to get rapid feedback, however synchronising with Vaclav and Ivor (in the Budapest office) is cumbersome — Skype is not ideal for pair programming. It takes the team the rest of the morning until Angela and Vaclav eventually find and fix the root cause of the fault — there was a null check missing in the initialisation routine Angela added yesterday.

Purpose. This scene illustrates the real potential of fault localisation during continuous integration. Faults in integration tests rarely occur, but have a big impact because they are difficult to locate hence difficult to assign to an individual. Moreover, software engineers must analyse code written the day before and integration tests written by others: the mental overhead of such context switches is significant. Finally, since these faults block all progress, team members must drop all other tasks to fix the build.

Scene 3: Retrospective. At the end of the sprint, Alex —the SCRUM master of the team– raises an issue during the retrospective meeting. He collected some statistics from the last years of development and found out that they experienced 394 integration faults for the 132 sprints, thus on average 3 integration faults per sprint. Consulting the issue database, he discovered that resolving one such integration fault takes on average 3.5 working hours. Knowing that these integration faults block all progress in the team, he estimates that broken builds for integration tests cost the team $+-1000$ person hours per year (8 sprints per year $*$ 3 faults per sprint $*$ 3.5 hours per fault $*$ 12 persons). Thus the team spends

just under 126 working days or —assuming 220 working days per year— 4% of its capacity on fixing integration faults.

Purpose. This scene illustrates the impact of such integration faults on the team productivity. Even if they occur rarely, the fact that it is an "all hands on deck" situation implies that resolving them takes a lot of effort. Even a small reduction in the time needed to fix an integration fault implies a significant gain in team productivity.

Additional material that was excluded from the original paper due to space constraints.

3.3.1 Requirements

From the above scenario, we can infer a few requirements that should hold for a fault localisation heuristic integrated in a continuous integration server.

Method Level Granularity. The seminal work on raw spectrum analysis (named Tarantula) was motivated by supporting an individual test engineer, and chose statement level granularity [11]. However, for fault localisation within integration tests, method level granularity is more appropriate. Indeed, the smallest element under test in object oriented testing is a method [78]. This also shows in modern IDE, where failing tests and stack traces report at method level. Last but not least, objects interact through methods, thus integration faults appear when objects don't invoke the methods according to the (often implicit) protocol.

Top 10. A fault localisation heuristic produces a ranked list of program elements likely to be at fault, thus the obvious question is how deep in the ranking the correct answer should be to still be considered acceptable. In the remainder of the chapter we set the threshold to 10, inspired by earlier work from Lucia et. al [68]. 10 is still an arbitrary number but was confirmed to be a good target during our interviews with real developers.

Fault localisation is applicable for complex systems with multiple branches and staged testing. Faults in integration tests in particular are very relevant: they seldom occur, but when they do, they have a big impact on the team productivity. Thus, to compare raw spectrum analysis against patterned spectrum analysis we should treat integration tests differently than unit tests.

3.4 PATTERNED SPECTRUM ANALYSIS

As explained earlier, current raw spectrum analysis heuristics comprise several variants, typically classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T*, and Naish2). In this chapter, we explore the hit-spectrum as a third dimension, incorporating patterns of method calls extracted by means of frequent itemset mining.

Below, we explain the details of the patterned spectrum analysis variant. We run the test suite and for each test case, collect the trace (Cf. Section 3.4.1), slice the trace into individual method traces (Cf. Section 3.4.2), reduce the sliced traces into call patterns for a method (Cf. Section 3.4.3), calculate the hit-spectrum by incorporating frequent itemset mining (Cf. Section 3.4.4), and finally rank the methods according to their likelihood of being at fault (Cf. Section 3.4.5).

3.4.1 Collecting the Trace

We maintain a single trace per test case. When a test runs, it invokes methods in the project base code. We intercept all the method calls originating from the base code method. We do not intercept calls in test methods, since we assume that the test oracles themselves are correct. The trace is collected by introducing logger functionality into the base code via AspectJ[2]. More specifically, we use a method call join point with a pointcut to pick out every call site. For each intercepted call, we collect the called method identifier, caller object identifier, and the caller method identifier. These identifiers are integers uniquely associated with a method name.

Listing 3.1: Code snippet for a sample method

```
1  public class A {
2   B objB;
3   C objC;
4   .....
5   public void collaborate() {
6    b.getData();
7    while(...) {
8     if(...)
9      c.getAttributes();
10    if(...)
11     c.setAttributes(...);
12    if(...)
13     c.processData(...);
14   } // while
15   b.saveData();
```

[2] http://www.eclipse.org/aspectj/

```
16    } // method
17  }
```

Table 3.3: A Sample Trace Highlighting Calls in Listing 3.1

Caller object id	Caller id†	Callee id‡
1	5	6
1	5	9
1	5	11
1	5	15
2	5	6
2	5	11
2	5	13
3	5	6
3	5	11
3	5	15
...	...	...

† caller id 5 indicates method collaborate() in Listing 2.1

‡ callee id is the line number in Listing 2.1

As an example, assuming the test case instantiates three objects of class A and calls method collaborate() (Listing 3.1) for each instance. A sample trace in a test case, specifically highlighting the method calls originating from the collaborate() method in Listing 2.1, is shown in Table 3.3. The three instances of class A are shown (id 1, 2, and 3) which each received a separate call to collaborate(). The execution of collaborate() on object id 1 resulted into a call to getData() (line 6), getAttributes() (line 9), setAttributes() (line 11), and finally saveData() (line 15). Execution of collaborate() on object id 2 and 3 results in a slightly different calls.

The 'caller object id' is the identifier of the caller object which calls the method, the 'caller' is the method from which the call is made and the 'callee' is the called method. When a method is executed in a class context (static methods can be executed without instantiating a class), there is no caller object, hence we mark the 'object caller id' as -1.

Considering the intercepted call getData() (line 6), the "caller object id" is the id of the class A object instantiated in the test case, the "caller id" is the id of method collaborate(), and the "callee id" is the id of method getData(). In a similar manner, calls originating from other methods such as method getData() of class B invoked from method collaborate() (line 6) are recorded in the trace.

3.4.2 Slicing the Trace

Once a trace for a test case is obtained, we slice the trace into individual method traces. Each sliced trace represents the trace for each executed method in the test case.

The sliced trace for a method m() in a test case $\mathcal{T}$ is represented as a set $\mathcal{T}_m = \{t_1, t_2, ..., t_n\}$, where t_i represents the method calls invoked from method m() through the same caller object. If the method m() is static, the calls appear in a single trace for 'caller object id' -1.

Referring to Table 3.3, $t_1 = \langle 6, 9, 11, 15\rangle$ for the calls of method collaborate() (id 5) with caller object id 1, $t_2 = \langle 6, 11, 13\rangle$ with caller object id 2, and $t_3 = \langle 6, 11, 15\rangle$ with caller object id 3. Therefore, the sliced trace τ_5 for method collaborate() (id 5) is as follows.

$$\mathcal{T}_5 = \{\langle 6, 9, 11, 15\rangle, \langle 6, 11, 13\rangle, \langle 6, 11, 15\rangle\} \tag{3.1}$$

3.4.3 Obtaining Call Patterns

We reduce the sliced trace $\mathcal{T}_m$ of a method $m()$ coming from a test case $\mathcal{T}$ into a set of call patterns $\mathcal{S}_{\mathcal{T}_m}$. To arrive at set of call patterns $\mathcal{S}_{\mathcal{T}_m}$, we adopt the *closed itemset mining algorithm* [23]. Given the sliced trace $\mathcal{T}_m$ of method m() in a test case $\mathcal{T}$, we define:

- $\mathcal{X}$ —*itemset*— a set of method calls.
- $\sigma(\mathcal{X})$ —support of $\mathcal{X}$— the number of t_i in $\mathcal{T}_m$ that contain this itemset $\mathcal{X}$.
- $minsup$ —minimum support of $\mathcal{X}$— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset $\mathcal{X}$ is frequent when $\sigma(\mathcal{X}) \geq minsup$.
- *closed itemset* — a frequent itemset $\mathcal{X}$ is closed if there exists no proper superset $\mathcal{X}'$ whose support is the same as the support of $\mathcal{X}$ (i-e. $\sigma(\mathcal{X}') = \sigma(\mathcal{X})$).

We refer to closed itemset $\mathcal{X}$ as a call pattern. We set $minsup$ to 1 to include call patterns for the methods executed with one object only or for those executed in a class context.

The set of call patterns $\mathcal{S}_{\mathcal{T}_m}$ for method collaborate() (id 5) from sliced trace $\mathcal{T}_5$ (Equation (3.1)) is as follows.

$$\mathcal{S}_{\mathcal{T}_5} = \{\{6, 9, 11, 15\}, \{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \tag{3.2}$$

3.4.4 Calculating the Hit-Spectrum

Unlike raw spectrum analysis, where there is a single test coverage matrix per program, patterned spectrum analysis creates a test coverage matrix for each executed method. In the raw spectrum analysis, a row of test coverage matrix corresponds to a method, which

is a program element per se, and the hit-spectrum (e_f, e_p, n_f, n_p) indicates whether or not the method is involved in test cases. In patterned spectrum analysis, there is a separate test coverage matrix for each method and a row corresponds to a call pattern (itemset $\mathcal{X}$) of the method. Here the call pattern ($\mathcal{X}$) is not a program element anymore. The hit-spectrum (e_f, e_p, n_f, n_p) of $\mathcal{X}$ not only indicates whether or not the method is involved in a test case, but also summarises its run-time behaviour.

The call patterns of a method m() in patterned spectrum analysis are obtained by running the set of failing test cases (denoted as $\mathbb{T}_F$) and the set of passing test cases (denoted as $\mathbb{T}_P$). We obtain a set of call patterns $\mathcal{S}_m$ (Equation (3.3)) for each method m() — which is the union of (i) the call patterns of a method resulting from the failing test cases ($\mathcal{S}_{\mathcal{T}_m} \in \mathbb{T}_F$) and (ii) the call patterns resulting from the passing test cases ($\mathcal{S}_{\mathcal{T}_m} \in \mathbb{T}_P$).

$$\mathcal{S}_m = \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_F\} \cup \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_P\} \tag{3.3}$$

The set $\mathcal{S}_m$ (Equation (3.3)) is used to construct the test coverage matrix for a method.

As an example, consider the set of call patterns for $\mathcal{S}_{\mathcal{T}_5}$ (Equation (3.2)) of the method collaborate(). Assuming, this call pattern results from a failing test case it will end up in $\mathcal{T} \in \mathbb{T}_F$. However, the same method collaborate() is also executed in a passing test case (i-e $\mathcal{T} \in \mathbb{T}_P$) and will result in another set of call patterns, shown in Equation (3.4).

$$\mathcal{S}_{\mathcal{T}_5} = \{\{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \tag{3.4}$$

Then, the call pattern set $\mathcal{S}_5$ for the method collaborate() becomes the union of Equation (3.2) and Equation (3.4).

$$\mathcal{S}_5 = \{\{6, 9, 11, 15\}, \{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \tag{3.5}$$

The hit spectrum is then calculated for each call pattern in the call pattern set which ultimately results in a test coverage matrix for each method. As an example, we show the test coverage matrix for collaborate() in Table 3.4.

3.4.5 Ranking Methods

Based on the test coverage matrix of call patterns for each method, each pattern in the call pattern set $\mathcal{S}_m$ (Equation (3.3)) gets a suspiciousness score. This suspiciousness is calculated by using a fault locator [13, 60, 61]. Then, we set the suspiciousness of the method as the maximum suspiciousness of its constituting patterns.

Suspiciousness per call pattern. Each call pattern $\mathcal{X} \in S_m$ (Equation (3.3)) gets a

Table 3.4: An Example Test Coverage Matrix for Method collaborate()

Call pattern $\mathcal{X}$	Failing tests ($\mathcal{T} \in \mathbb{T}_F$)	Passing tests ($\mathcal{T} \in \mathbb{T}_P$)	$e_f(\mathcal{X})$	$e_p(\mathcal{X})$	$n_f(\mathcal{X})$	$n_p(\mathcal{X})$	w($\mathcal{X}$)
	t_1	t_2					
$\{6, 9, 11, 15\}$	1	0	1	0	0	1	1.0
$\{6, 11, 13\}$	1	1	1	1	0	0	0.7
$\{6, 11, 15\}$	1	1	1	1	0	0	0.7
$\{6, 11\}$	1	1	1	1	0	0	0.7

suspiciousness $W(\mathcal{X})$ calculated with a fault locator. In principle, any fault locator can be chosen from the literature. However, for our comparison purpose we tested all four fault locators mentioned in Table 3.2 in patterned spectrum analysis and Ochiai (Equation (3.6)) came out as the best performing one. For our running example, the suspiciousness $W(\mathcal{X})$ for each call pattern of method collaborate() is given in Table 3.4.

$$W(\mathcal{X}) = \frac{e_f(\mathcal{X})}{\sqrt{(e_f(\mathcal{X}) + n_f(\mathcal{X})) * (e_f(\mathcal{X}) + e_p(\mathcal{X}))}} \tag{3.6}$$

Suspiciousness per method. Each method m() gets a suspiciousness $W(m)$ which is the suspiciousness of the call pattern $\mathcal{X}$ with the highest suspiciousness (Equation (3.7)). We choose the maximum (instead of average) for the suspiciousness score because the technique is looking for exceptional traces: one unique and highly suspicious pattern is more important than several unsuspicious ones. Those methods without call patterns set have suspiciousness 0. The suspiciousness for method collaborate() $W(5)$ in our running example is 1.0, which is the suspiciousness of the call pattern ($\{6, 9, 11, 15\}$)— with highest suspiciousness (Table 3.4).

$$W(m) = \max_{\mathcal{X} \in S_m} \Big(W(\mathcal{X}) \Big) \tag{3.7}$$

Ranking. Finally, a ranking of all executed methods is produced using their suspiciousness W(m). The suspiciousness of the method indicates its likelihood of being at fault. Those methods with the highest suspiciousness appear a the top in the ranking.

3.5 CASE STUDY SETUP

Given the current state of the art (referred to as raw spectrum analysis) and the variant proposed in this chapter (referred to as patterned spectrum analysis), we can now compare the effectiveness of these two heuristics from the perspective of a continuous integration scenario. We give some details about the dataset used for the comparison (Defects4J), the evaluation metric (Wasted Effort), to finish with the research questions, and protocol driving the comparison.

Dataset. We use 351 real faults from 5 open source java projects: Apache Commons Math, Apache Commons Lang, Joda-Time, JFreeChart, and Google Closure Compiler. The descriptive statistics of these projects are reported in Table 3.5. These faults have been collected by Just et. al. into a database called Defects4J[3] (a **d**atabase of **e**xisting **f**aults to **e**nable **c**ontrolled **t**esting **s**tudies for **J**ava programs) [59]. The database contains meta info about each fault including the source classes modified to fix the fault, the test cases that expose the fault, and the test cases that trigger at least one of the modified classes. Although, the framework does not explicitly list the modified methods, we could reverse engineer those by means of the patches that come with the framework. Note that we excluded 3 faults of Apache Commons Lang, 2 faults of Apache Commons Math, and 1 fault of Joda-Time since the fault was not located inside a method.

Unfortunately, the Defects4J dataset does not distinguish between unit tests or integration tests. As argued in the Scenario (Section 3.3), this is a crucial factor when assessing a fault localisation heuristic in a continuous integration context. We, therefore, manually inspected a sample of test methods and noticed that four projects (Apache Commons Math, Apache Commons Lang, Joda-Time, and JFreeChart) mainly contain unit tests: they have a small (often empty) set-up method, and test methods contain only a few asserts. One project however (Closure Compiler) relies on integration tests. The test cases there, are a subclass of CompilerTestCase that defines a few template methods, which are the entry point to several classes in the base code of the project.

To corroborate this manual inspection, we calculated the number of methods triggered in each fault spectrum analysis. The assumption here is that integration tests exercise several methods in various classes, consequently the fault spectrum analysis should trigger many methods as well. Thus, projects which gravitate towards integration testing should trigger many methods while projects gravitating towards unit tests should trigger far fewer. The results are shown in the last two columns (μ and σ) of Table 3.5; listing the average and standard deviation per project respectively. The high number of μ for the Closure project is an indication that the Closure tests exercise a lot of the base code, yet the high standard deviation σ signals the presence of unit tests as well. On the other

[3] http://defects4j.org

Table 3.5: Descriptive Statistics for the Projects Used in Our Experiments — Defects4J

Project	# of Bugs	Source KLoC	Test KLoC	# of Tests	Age (years)	# Methods triggered (μ†)	# Methods triggered ($\sigma^{\ddagger}$)
Math[1]	106	85	19	3,602	11	153.1	140.8
Lang[2]	65	22	6	2,245	12	89.3	55.2
Time[3]	27	28	53	4,130	11	586.0	209.5
Chart[4]	26	96	50	2,205	7	306.9	407.5
Closure[5]	133	90	83	7,927	5	2043.0	1228.9

† Average number of methods triggered by the Spectrum based fault localisation—— ‡ Standard deviation

([1]) Apache Commons Math – `http://commons.apache.org/math`

([2]) Apache Commons Lang – `http://commons.apache.org/lang`

([3]) Joda-Time – `http://joda.org/joda-time`

([4]) JFreeChart – `http://jfree.org/jfreechart`

([5]) Google Closure Compiler – `http://code.google.com/closure/compiler/`

hand, the low number of μ for the other project hints at mostly unit tests, yet Chart has a standard deviation σ of 407 (compared to an average of 306), indicating a few outlier tests which cover a lot of the base code.

The Defects4J dataset does not distinguish between unit tests or integration tests. However, one project (Closure Compiler) gravitates towards integration tests. Therefore, the results of the Closure Compiler should serve as circumstantial evidence during the comparison.

Wasted Effort. As mentioned earlier, we compare by means of the *wasted effort* metric, commonly adopted in recent research [60, 61, 70]. The wasted effort indicates the number of non-faulty methods to inspect in vain before reaching the faulty method.

$$\text{wasted effort} = m + (n+1)/2 \tag{3.8}$$

Where

- m is the number of non-faulty methods ranked strictly higher than the faulty method;
- n is the number of non-faulty methods with equal rank to the faulty method. This deals with ties in the ranking.

The comparison is driven by the following research questions.

RQ1. *Which ranking results in the lowest wasted effort: raw spectrum analysis or patterned spectrum analysis?*

Motivation. This is the first step of the comparison; assessing which of the two fault localisation methods provides the best overall ranking.

RQ2. *How often do raw spectrum analysis and patterned spectrum analysis rankings result in a wasted effort $\leq$ 10?*

Motivation. Based on the scenario (Section 3.3), we investigate how many times the location of the fault is ranked in the first 10 items.

RQ3. *How does the number of triggered methods affect the wasted effort of raw spectrum analysis and patterned spectrum analysis?*

Motivation. Again, based on the scenario (Section 3.3) we gauge the impact of integration tests. The number of methods triggered by the fault spectrum analysis acts as a proxy for the degree of integration tests in the test suite.

Fault Locator. One dimension of variation in spectrum based fault localisation is the fault locator; Table 3.2 lists the most popular ones. As explained in Section 3.4.5, for comparison purpose we use Ochiai for patterned spectrum analysis. However, for the optimal

Table 3.6: Naish within Raw Spectrum Analysis vs. Tarantula, Ochiai and T*

Faul Locator	$<$	$>$	$=$
Tarantula	50 (38%)	8 (6%)	75 (56%)
Ochiai	46 (35%)	6 (5%)	81 (61%)
T*	20 (15%)	4 (3%)	109 (82%)

configuration of raw spectrum analysis, we actually tested all four fault locators (Table 3.2). Naish2 performed the best on the Defects4J dataset with method level granularity as can be seen in Table 3.6. There, we compare the wasted effort of Naish2 against the wasted effort of other fault locators, using the 133 defects in the Closure project. For most defects, Naish2 results in a better or equal ranking; only for a few defects is the ranking with other locators better. For space reasons we do not show the comparison on other projects, but there as well Naish2 was the best. Hence, we choose Ochiai for patterned spectrum analysis and Naish2 for raw spectrum analysis in the case study.

Protocol. To run the fault spectrum analysis, we check out a faulty version (V_{fault}) for each project. Then we run the actual spectrum based fault localisation for all *relevant* test cases, i.e. all test classes which trigger at least one of the source classes modified to fix the fault as recorded in the Defects4J dataset. Given the continuous integration context for this research, this is the most logical way to minimise the number of tests which are fed into the spectrum based fault localisation. Note that this explains why the number of methods triggered by a fault spectrum is a good indicator for the integration tests; since the tests are chosen such that they cover all changes made to fix the defect.

3.6 RESULTS AND DISCUSSION

In this section, we address the three research questions introduced in Section 3.5. This allows for a multifaceted comparison of the effectiveness of patterned spectrum analysis against the state of the art raw spectrum analysis.

RQ1. *Which ranking results in the lowest wasted effort: raw spectrum analysis or patterned spectrum analysis?*

To determine the best performing heuristic, we plot the wasted effort for all of the faults for both heuristics. To allow for an easy exploration of the nature of the difference, we sort the faults according to the wasted effort of raw spectrum analysis and plot the

Table 3.7: Comparing Wasted Effort: Patterned Spectrum Analysis vs Raw Spectrum Analysis

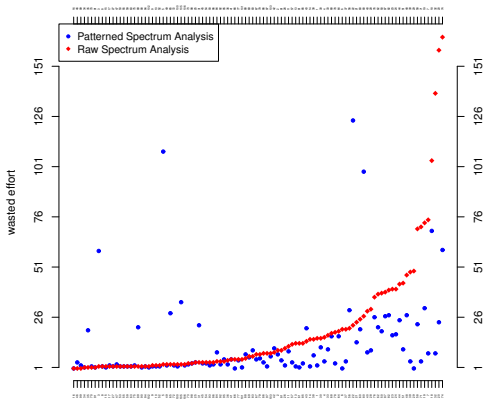
Project	$<$	$>$	$=$	Total
Math	69 (66%)	22 (21%)	13 (13%)	104
Lang	36 (58%)	14 (23%)	12 (19%)	62
Time	16 (62%)	7 (27%)	3 (12%)	26
Chart	16 (62%)	7 (27%)	3 (12%)	26
Closure	101 (76 %)	30 (23 %)	2 (2%)	133
Total	238 (68 %)	80 (23 %)	33 (9%)	351

wasted effort for patterned spectrum analysis accordingly. The result can be seen in (Figures 3.1a, 3.1b, 3.1c, 3.1d, and 3.1e). Next, we count all the faults for which the wasted effort (in patterned spectrum analysis) is strictly less ($<$), strictly more ($>$), or the same ($=$) and list the absolute numbers per project (See Table 3.7).

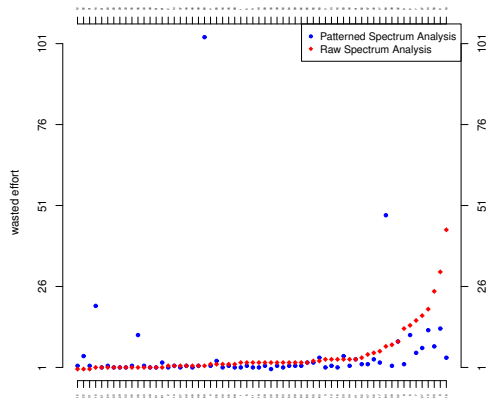
To illustrate how the rankings of the heuristics differ, we inspect fault 40 of the Closure project where the wasted effort for patterned spectrum analysis is 0.5 (the faulty method is ranked first), while for raw spectrum analysis the wasted effort is 183. This is due to the fact that the faulty method has a call pattern which is unique in all failing test cases, hence is easily picked up by patterned spectrum analysis. On the other hand, just marking whether

Table 3.8: Number of Faults where Wasted Effort is ≤ 10

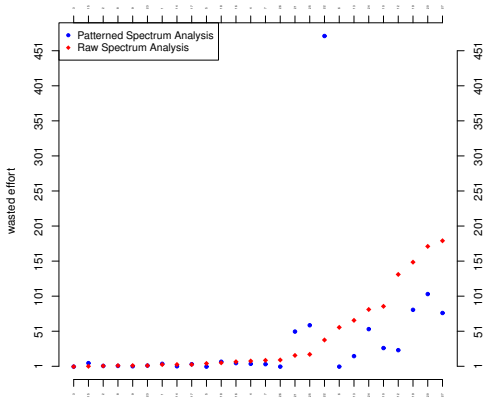
Project	Patterned Spectrum Analysis (PSA)	Raw Spectrum Analysis (RSA)	PSA - RSA	Total
Math	73 (70%)	59 (57%)	14	104
Lang	55 (89%)	54 (87%)	1	62
Time	16 (62%)	14 (54%)	2	26
Chart	16 (62%)	13 (50%)	3	26
Closure	56 (42%)	30 (23%)	26	133
Total	216 (62%)	170 (48%)	46	351



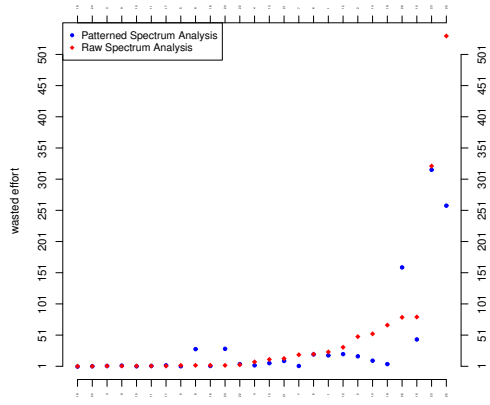
(a) Math



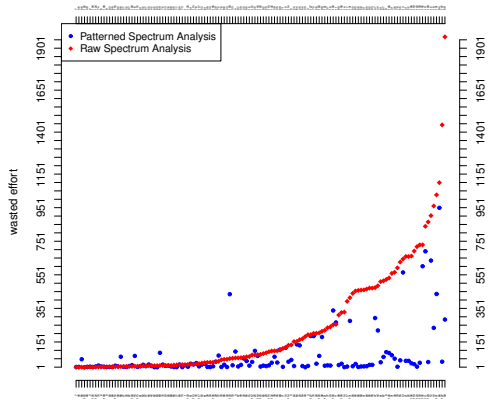
(b) Lang



(c) Time



(d) Chart



(e) Closure

Figure 3.1: The comparison plots of all the rankings in each Lang

Table 3.9: Number of Triggered Methods vs. Wasted Effort

Bin	PSA†			RSA‡		
	Q1	Median	Q3	Q1	Median	Q3
4-43	1.0	1.5	2.5	1.0	1.8	2.9
44-71	1.5	3.0	6.8	2.2	2.8	8.5
72-91	1.5	2.8	9.1	2.4	5.2	13.0
92-134	1.5	2.8	11.5	1.5	3.8	17.6
137-202	1.5	3.2	9.1	1.5	3.2	15.5
204-397	2.0	8.0	23.5	3.5	20.0	73.0
423-892	1.9	5.0	51.4	3.5	9.0	70.8
917-1262	5.8	14.0	38.5	10.4	263.0	511.6
1273-1721	8.2	20.8	56.4	33.9	97.8	203.1
1752-2464	2.5	11.2	40.9	12.4	50.0	196.0
2523-5825	5.0	24.0	77.5	11.0	115.5	561.1

or not the method is executed, is not discriminating in raw spectrum analysis. The number of failing test cases covering the faulty method and non-faulty methods, is the same 169. Yet, the non-faulty methods have more suspiciousness than faulty method because the number of passing test cases covering the non-faulty methods is less. Since more passing test cases cover the faulty method (high value of e_p), it renders the faulty method less suspicious.

For 68% faults in the dataset, the wasted effort with patterned spectrum analysis is lower than raw spectrum analysis. Moreover, this improvement is a lot better for the Closure project (the one system in the data set which gravitates towards integration tests), where we see an improvement for 76% of faults (101 out of 131).

RQ2. *How often do raw spectrum analysis and patterned spectrum analysis rankings result in a wasted effort ≤ 10?*

Inspired by the scenario in Section 3.3, we count how many times the location of the fault is ranked in the top 10. To deal with ties in the ranking (especially at position 10),

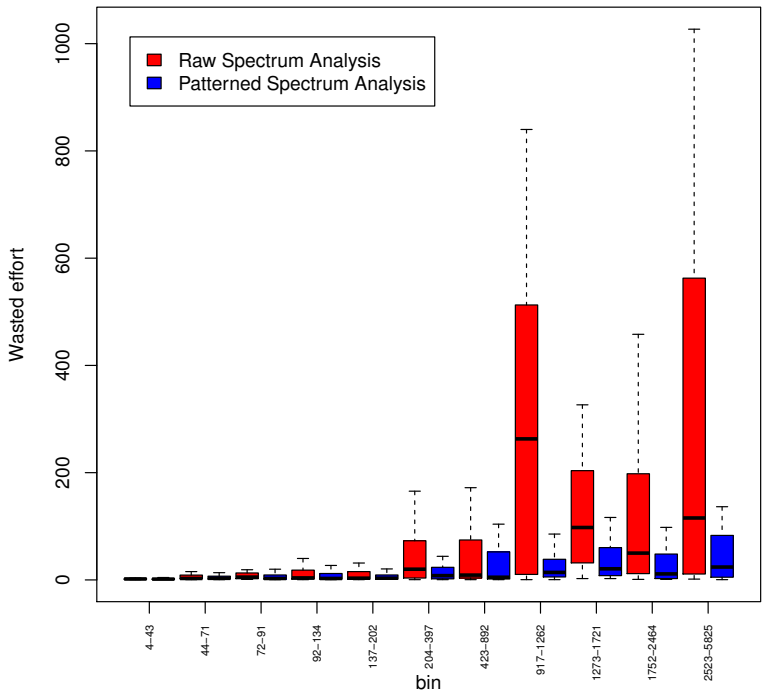


Figure 3.2: Number of Triggered Methods vs. Wasted Effort

we identify these as having a wasted effort ≤ 10.

Table 3.8 shows, for each project, the number of faults where the wasted effort is within the range of 10 with both heuristics. For three projects (Lang, Time, and Chart), the performance of the patterned spectrum analysis is comparable but still better than the one of the raw spectrum analysis. Whereas, for the remaining two projects (Math and Closure) the performance of the patterned spectrum analysis is noticeably better. These findings confirm that patterned spectrum analysis ranks more faults in the top 10. However, there are still a large amount of faults where the ranking is poor (wasted effort $>$ 10). Especially, for the Closure project less than half (42%) of the faults are ranked in the top 10. Hence, there is still room for improvement, which we will cover in Section 3.7.

The patterned spectrum analysis succeeds in ranking the root cause of the fault in the top 10 for 62% of the faults, against 48% for raw spectrum analysis.

RQ3. *How does the number of triggered methods affect the wasted effort of raw spectrum analysis and patterned spectrum analysis?*

In Section 3.5, we argued that the number of methods triggered by the fault spectrum analysis is an indicator of the gravitation towards integration tests (see also the last two columns in Table 3.5). If that is the case, a good spectrum based fault localisation heuristic should obtain a good ranking for a particular fault regardless of the number of triggered methods. Again, based on the scenario (Section 3.3), we gauge the impact of integration tests.

Therefore, for each fault, we calculate the number of methods triggered by the fault spectrum analysis. We then sort the faults according to the number of methods and inspect the trend with respect to the number of triggered methods. Unfortunately, the standard deviation for the number of triggered methods is really high (see the σ column in Table 3.5) and a normal scatterplot mainly showed the noise. Therefore, we group the faults according to the triggered methods into 11 bins of 32 elements. (As these numbers did not divide well, there were two bins having 30 and 33 triggered methods respectively.) This binning was decided as a trade-off for having an equal number of elements per bin and enough bins to highlight a trend in the number of triggered methods, if any. For each of the bins, we calculated the first quartile, median, and the third quartile, listing them all in Table 3.9 and plotting them in a series of boxplots (Figure 3.2)

Table 3.9 and Figure 3.2 illustrate that the number of methods triggered has little effect on patterned spectrum analysis, however, quite a lot on raw spectrum analysis. The last four bins, in particular, contain faults which trigger more than thousand methods. The median wasted effort for patterned spectrum analysis is four to eighteen times lower than raw spectrum analysis.

The better rankings for Closure in Table 3.7 and Table 3.8 are inconclusive, as one case is not enough to generalise upon. Yet, based on an analysis of the number of methods triggered by the fault spectrum, there is at least circumstantial evidence that patterned spectrum analysis performs better for integration tests.

3.7 POSSIBLE IMPROVEMENTS

Upon closer inspection of those faults ranked high by the patterned spectrum analysis heuristic, we can infer some suggestions for improvement regarding future variations.

First of all, an inherent limitation is that a faulty method which does not call any other methods will always be ranked at the bottom. Indeed, such methods don't have a call

pattern (which is the primary coverage element appearing in the test coverage matrix), thus the method gets suspiciousness 0. In our case study, we noticed a few cases where none of the faulty methods had any call pattern. More specifically, there are 4 such cases in the Math project, 3 in the Chart project, 2 in the Time and Lang projects, and only 1 in the Closure project. The best example corresponds to the highest wasted effort on fault 60 of the Lang project (See Listing 3.2). Indeed, the faulty method contains(char) in class org.apache.commons.lang.text.StrBuilder gets suspiciousness 0 because the for loop only performs direct accesses to memory and never calls any methods.

Similarly, the highest wasted effort for fault 22 in the Math project is due to the faulty method isSupportUpperBoundInclusive() in class distribution.UniformRealDistribution which again never calls any other methods. In this case, the method body contained a single statement return false;; the bug fix replaced it by return true;. A last example is fault 22 in Time project; where the fault resided in a faulty constructor, hence did again not have any method call pattern.

Listing 3.2: Code snippet for a sample method

```
1  public boolean contains(char ch) {
2   char[] thisBuf = buffer;
3   // Correct code
4  //for (int i = 0; i < this.size; i++) {
5   // Incorrect code
6  for (int i = 0;i < thisBuf.length;i++) {
7    if (thisBuf[i] == ch) {
8     return true;
9    }
10  }
11 }
```

Listing 3.3: Unique call sequence in faulty method tryMinimizeExits(Node,int,String)

```
1  Node.getLastChild()
2  NodeUtil.getCatchBlock(Node)
3  NodeUtil.hasCatchHandler(Node)
4  NodeUtil.hasFinally(Node)
5  Node.getLastChild()
6  tryMinimizeExits(Node,int,String)
```

Second, patterned spectrum analysis is often able to push the faulty method high in the ranking, however there are several cases where it never reaches the top 10. A nice example is fault 126 in Closure, where the wasted effort for patterned spectrum analysis is 85.5. This value is still lower than the one given by raw spectrum analysis (532.5), yet it is too high to ever be considered in a realistic scenario. Manually analysing the traces of

the faulty method tryMinimizeExits(Node,int,String) in class com.google.javascript.jscomp.MinimizeExitPoints, we found a unique call pattern (Listing 3.3) which is only called in the failing tests. The bug fix[4] reveals that the developers removed the “if check" with a finally block. This “if check" involves the last 3 calls in Listing 3.3 (lines 4-6). Despite being unique, the reason why this call pattern was not picked up by patterned spectrum analysis is because the order of method calls is crucial. Indeed, the call pattern in patterned spectrum analysis is an itemset, hence the call pattern is not order preserving and has no repetitive method calls. Note that the importance of the call-order was also pointed out by Lo et al. [79].

As a future improvements of patterned spectrum analysis, we might incorporate statements or branches into the hit-spectrum. The call-order of methods, as well, is relevant information to incorporate into the hit-spectrum.

3.8 THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [80, 81]), we organise them into four categories.

Construct validity – do we measure what was intended ?

Wasted Effort. In this research, we adopted the wasted effort metric to compare raw spectrum analysis against patterned spectrum analysis. However, in information retrieval rankings where users do not want to inspect all outcomes other measures are considered, such as Mean Reciprocal Rank (MRR) or Mean Average Precision (MAP) [10, 82]. It is unclear whether the use of these relative evaluation metrics would alter the results. Nevertheless, the use of an absolute metric alleviates other concerns [61, 71]. Therefore, the impact is minimal.

Fault Masking. One particular phenomenon which occurs in a few faults in the Defects4J dataset is “fault masking" [61]. This is a fault which is spread over multiple locations and where triggering one location already fails the test. The fix for fault 23 of project Chart for instance, comprises two changes in two separate methods of the class “renderer.category.MinMaxCategoryRenderer ". The first change is to override “equals(Object) " method and the second involves changes in method “setGroupStroke(- Stroke) ". The test case which exposes the defect calls both methods, yet the test case fails on the first as-

[4] https://github.com/google/closure-compiler/commit/bd2803

sertion calling the "equals(Object) " method thereby masking the "setGroupStroke(Stroke) " method. The question then is what a fault localisation should report: one location or all locations ? Furthermore, how should we assess the ranking of multiple locations. In this research, inspired by earlier work [10, 82], we took the assumption that reporting one location is sufficient and use the highest ranking of all possible locations. However, one could make other assumptions.

Internal validity – are there unknown factors which might affect the outcome of the analyses ?

Multiple faults. One often heard critique on fault localisation heuristics in general and spectrum based fault localisation in particular is that when multiple faults exist, the heuristic will confuse their effects and its accuracy will decrease. Two independent research teams confirmed that multiple faults indeed influence the accuracy of the heuristic, however it created a negligible effect on the effectiveness [62, 83]. We ignore the potential effect of multiple faults in this chapter. Nevertheless, future research should study the effect of multiple faults.

Correctness of the Oracle. The continuous integration scenario in Section 3.3 makes the assumption that the test oracle itself is infallible. However this does not hold in practice: Christophe et. al. observed that functional tests written in the Selenium library get updated frequently [84]. We ignore the effects of the tests being at fault in this chapter, but here as well point out that this is something to be studied in future work.

External validity – to what extent is it possible to generalise the findings ? In our study, we experimented with 351 real faults drawn from five representative open source object oriented projects from Defects4J dataset; the most recent defect dataset currently available. Obviously, it remains to be seen whether similar results would hold for other defects in other systems. In particular, there is a bias towards unit test in the Defects4J dataset, with only the Closure project gravitating towards integration tests. Further research is needed to verify whether the patterned spectrum analysis is indeed a lot better on integration tests in other systems.

Reliability – is the result dependent on the tools ? All the tools involved in this case study (i.e. creating the traces, calculating the raw spectrum analysis, and patterned spectrum analysis rankings) have been created by one of the authors. They have been tested over a period of 2 years; thus the risk of faults in the tools is small. Moreover, for the calculation of the raw spectrum analysis rankings we compared as best as possible against the results reported in earlier papers. The algorithm for frequent itemset mining was adopted from open source library SPMF[5], hence there as well the risk of faults is small.

[5] http://www.philippe-fournier-viger.com/spmf/

3.9 CONCLUSION

Spectrum based fault localisation is a class of heuristics known to be effective for localising faults in existing software systems. These heuristics compare execution traces of failing and passing test runs to produce a ranked list of program elements likely to be at fault. The current state of the art (referred to as raw spectrum analysis) comprises several variants, typically classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T*, and Naish2). In this chapter, we explore a third dimension: the hit-spectrum. More specifically, we propose a variant (referred to as patterned spectrum analysis) which extends the hit-spectrum with patterns of method calls extracted by means of frequent itemset mining.

The motivation for the patterned spectrum analysis variant stems from a series of contacts with software developers working in Agile projects and relying on continuous integration to run all the tests. Complex systems with multiple branches and staged testing could really benefit from fault localisation. Faults in integration tests, in particular, are very relevant: they seldom occur, but if they do, they have a big impact on the team productivity.

Inspired by the continuous integration motivational example, we compare patterned spectrum analysis against raw spectrum analysis using the Defects4J dataset. This dataset contains 351 real faults drawn from five representative open source java projects. Despite a bias towards unit tests in the dataset, we demonstrate that patterned spectrum analysis is more effective in localising the fault. For 68% faults in the dataset, the wasted effort with patterned spectrum analysis is lower than raw spectrum analysis. Also, patterned spectrum analysis succeeds in ranking the root cause of the fault in the top 10 for 63% of the defects, against 48% for raw spectrum analysis. Moreover, this improvement is a lot better for the Closure project; the one system in the data set which gravitates towards integration tests. There, we see an improvement for 76% defects (101 out of 131). The better rankings for Closure are inconclusive (one case is not enough to generalise upon), yet based on an analysis of the number of methods triggered by the fault spectrum, there is at least circumstantial evidence that patterned spectrum analysis performs better for integration tests. Despite this improvement, we collect anecdotal evidence from those situations where the patterned spectrum analysis ranking is less adequate and derive suggestions for future improvements.

3.10 ACKNOWLEDGMENTS

Thanks to prof. Martin Monperrus for reviewing an early draft of this chapter. This work is sponsored by (i) the Higher Education Commission of Pakistan under a project titled “Strengthening of University of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled “Change-centric Quality Assurance (CHAQ)” with number 120028.

Chapter 4

On the Use of Sequence Mining within Spectrum Based Fault Localisation

On the Use of Sequence Mining within Spectrum Based Fault Localisation
Gulsher Laghari and Serge Demeyer
In *Proceedings of the Symposium on Applied Computing (SAC 2018)*, 1916–1924. Pau, France. April, 2018.
DOI: https://doi.org/10.1145/3167132.3167337.

This chapter was originally published in the *Proceedings of the Symposium on Applied Computing (SAC 2018).*

ABSTRACT

Spectrum based fault localisation is a widely studied class of heuristics for locating faults within a software program. Unfortunately, the current state of the art ignores the inherent dependencies between the methods leading up to the fault, hence having a limited diagnostic accuracy. In this chapter we present a variant of spectrum based fault localisation, which leverages series of method calls by means of sequence mining. We validate our variant (we refer to it as sequenced spectrum analysis*) on the* Defects4J *benchmark, demonstrating that* sequenced spectrum analysis *gains a 12% points improvement against the state of the art.*

4.1 INTRODUCTION

Software defects remain a primary concern within the software engineering community [43]. Since the source code provides the ultimate description of the behaviour of

the system, it is there that software engineers search for the the root cause of a defect— the so-called *fault*— and fix it subsequently. Fault localisation is widely acknowledged to be one of the more difficult and time consuming steps while fixing defects and it is, therefore, a heavily investigated research topic [14].

In this chapter, we focus on what is known as *spectrum based fault localisation* [11, 60, 67]. Spectrum based fault localisation, a lightweight automated fault localisation technique, statistically compares executions traces from both failing and passing test cases to pinpoint a faulty program element. It produces a ranked list of program elements, indicating the likelihood of a program element being at fault. Spectrum based fault localisation comprises three steps: (i) creating a *test coverage matrix*; (ii) deducing the *hit-spectrum*, and (iii) applying a *fault locator*.

Today, most variants of spectrum based fault localisation focus on step (iii), and experiment with different fault locator functions (e.g., Ochiai [13] and Naish2 [17]). Recently, however, a new branch of research investigated variations of step (ii) providing alternative ways for deducing the hit-spectrum (e.g., time spectra [19], frequent itemsets [30, 31], and method invariants [20]). This chapter investigates one other alternative for deducing the hit-spectrum, namely *sequence mining*. As such, we make the following contributions.

1. We present a variant of spectrum based fault localisation (referred to as *sequenced spectrum analysis* in the remainder of this chapter) which leverages series of method calls by means of sequence mining.

2. We use 47 known fault locators to create a suite of spectrum based fault localisation heuristics— the current state-of-the-art (referred to as *raw spectrum analysis*)— and evaluate them on real faults. This sets the baseline for the comparison.

3. We compare sequenced spectrum analysis against the raw spectrum analysis using the Defects4J dataset [59].

4. We use several evaluation metrics during that comparison, effectively adhering to the concerns of absolute measure [61, 71], early precision [8], and total recall [8] which implies a stringent comparison.

The remainder of this chapter is organised as follows. Section 4.2 lists the known variants of spectrum based fault localisation for comparison, while Section 4.3 explains the details of the variant proposed here. Section 4.4 describes the set-up of the comparison, which naturally leads to Section 4.5 reporting the results. After a discussion on the related work in Section 4.6 and the threats to validity in Section 4.7, we come to a conclusion in Section 4.8.

4.2 BACKGROUND

Spectrum based fault localisation takes as input the faulty program and a test-suite where at least one test case exposes the defect. It produces, as the output, a ranked list of program elements where the most suspicious program elements appear at the top of the list.

There are several concerns to take into account when applying spectrum based fault localisation on a faulty program. First, it involves the decision to select a *granularity* level of a program element. The choices for granularity include from fine-grained statements to course-grained classes.

Second, the coverage of selected program element is collected by running the test-suite for the faulty program. This coverage is organised into a data structure called the *test coverage matrix*. The rows in this matrix correspond to elements under test and the columns represent the test cases. Each cell in the matrix marks whether a given element under test is executed by the test case (marked as 1) or not (marked as 0).

Third, the test coverage matrix is summarised into the *hit- spectrum*— a summarised abstract behaviour of the program. The hit-spectrum of an element under test is a tuple of four values (e_f, e_p, n_f, n_p). Where e_f and e_p are the numbers of failing and passing test cases that execute the element under test respectively and n_f and n_p are the numbers of failing and passing test cases that do not execute the element under test respectively.

Fourth, the *fault locator function* translates the hit-spectrum into suspiciousness of the element under test. This suspiciousness, which is function of the fault locator, indicates the likelihood of the element under test to be at fault. Most fault locator functions have a range in interval [0, 1]. Thus, the suspiciousness value for an element under test may have the lowest value 0 (not suspicious at all) to 1 (highly suspicious). The underlying intuition is that an element under test which is executed more in failing tests and less in passing tests gets a higher suspiciousness and appears at the top location in the ranked list. Sorting the elements under test according to their suspiciousness in descending order produces the ranked list. We refer to this kind of fault localisation analysis as *raw spectrum analysis*.

When applying spectrum based fault localisation, there are three avenues to improve the effectiveness of the heuristic. The first is exploring the different levels of granularity of program elements. The state of the art has almost explored all possible granularity levels from *statements* [11, 64, 65, 85], *blocks* [16, 67, 68, 69], *methods* [20, 31, 60, 70], and to *classes* [22, 30]. In this chapter, we select method-level granularity for four reasons. (1) In object oriented testing a method is the smallest element under test [78]. (2) Objects interact through methods by following a certain protocol on the calling sequence of its

methods [56]. For complicated protocols this is a source of subtle bugs which are notoriously difficult to resolve [86]. (3) A method often provides sufficient context needed to help developers understand a bug [20]. (4) Developers expressed a slight preference for method-level granularity [87].

The second dimension is to optimise the fault locator function which was untill now the primary avenue for improvement. The efforts include applying functions used in the molecular biology domain for fault localisation [67], applying association measures from data mining for fault localisation [16], proposing fault locators based on a theoretical model [17], and evolving a completely new set of fault locators through genetic programming [18].

The third dimension, which has remained largely unexplored up until now, is to try a variation of the hit-spectrum— the input to the fault locator. Yilmaz et al. for example adopted traces of method execution times instead of a mere count of passing and failing tests [19]. Dallmeier et al. extracted sequence of method calls by sliding a window over execution traces of classes to identify faulty classes [22]. Similarly, we adopted itemset mining to pinpoint faulty classes [30]. Later, we explored a variant of the hit-spectrum adopting frequent itemset mining to localise faulty methods [31]. In this chapter, we present another variant of the hit-spectrum adopting sequence mining to locate faulty methods.

The motivation for exploring sequence mining in the hit-spectrum is that in the traditional fault localisation, the hit spectrum only tells whether or not the method is involved in a test case. However, it ignores the inherent dependencies between the calls leading up to the fault. In particular, branch conditions, data inputs, or exceptions thrown may be the real cause for deviating behaviour of the failing test [22]. Since fault localisation has access to the complete call trace anyway, it is relatively easy to incorporate information about the call sequences itself. Consequently, we modify the hit-spectrum by adopting sequence mining, referring to this kind of fault localisation analysis as *sequenced spectrum analysis* compared to the traditional approach referred to as *raw spectrum analysis*.

> *The hit-spectrum in raw spectrum analysis ignores the inherent dependency relationships between the calls leading up to the fault. In sequenced spectrum analysis, we modify the hit-spectrum by incorporating series of method calls mined from the execution traces.*

4.3 SEQUENCED SPECTRUM ANALYSIS

Here, we briefly describe the steps in our sequenced spectrum analysis. We run the test cases on a faulty program and in each test case, (1) collect the traces for each executed method of the project (Section 4.3.1), (2) mine the call sequences from these traces (Section 4.3.2), (3) calculate the hit-spectrum (Section 4.3.3), and finally (4) rank the methods (Section 4.3.4) according to their likelihood of being at fault.

4.3.1 Collecting the Trace

In each test case, during the execution of a method, we intercept each method call directly originating from the method and record it into the trace. The intercepted call can be a call to a project method or to the external library method. Note that we incorporate calls to the constructors, hence have knowledge about the creation of objects as well.

A trace is collected by introducing the logger functionality into the base code via AspectJ[1]. More specifically, we use method execution and method call join points. For a method execution join point, there are two advices (before and after), while there is one advice for method call join point. In the *before execution* advice, a trace is initiated for the executed method. In the *before call* advice, which picks out every call site, we collect the names of both callee and the caller method and add the called method into the trace of the caller method. Finally, in the *after execution* advice, the current trace for the method is closed. To save the memory, we assign unique integer identifier to each executed method and add the identifier to the trace instead of the name.

As a method can also execute one or more times in a test case, we separate the call traces for each execution. Thus, the complete trace for a method m() in a test case $\mathcal{T}$ is represented as a set $\mathcal{T}_m = \{t_1, t_2, ..., t_n\}$. Where t_i is a list of the method calls invoked directly from method m() during its i^{th} execution. This implies that the calling relation is not followed transitively but is terminated after one level. If method m() in Listing 4.1 executed twice in a test case $\mathcal{T}$, its trace would be $\mathcal{T}_m = \{\langle m2, m3\rangle, \langle m2, m3\rangle\}$.

Listing 4.1: Example method

```
1  public void m(){
2   m2();
3   m3();
4  }
```

[1] http://www.eclipse.org/aspectj/

4.3.2 Obtaining Call Sequences

Once the call traces are collected, we mine the sequence of method calls from the traces of a method. Normally, a method executes only once in a test case resulting into only a single trace for a method ($|\mathcal{T}_m| = 1$).

Hence, we adopt the *MARBLES algorithm* to mine the call sequences from the method call traces [88]. This algorithm mines general, parallel, and serial episodes (subsequences) from a large sequence (*a single call trace in our case*) sliding a window of fixed size. Since we are interested in an order-preserving sequence of method calls, we only use the serial episodes. From here on, we refer to these episodes simply as sequences.

In this experiment, we use a window size of 8. We restrict the window size to 8 inspired by Dallmeier et. al. who found that increasing the window size beyond 8 did not increase effectiveness of fault localisation [22]. Also for window sizes greater than 8 when applied on long traces, MARBLES takes a long time to produce a result. We apply the algorithm to translate the complete trace $\mathcal{T}_m$ for a method m() obtained in a test case $\mathcal{T}$ into a set of call sequences $S_{\mathcal{T}_m}$. Thus, for each call trace $ti \in \mathcal{T}_m$ as input, the algorithm produces s_i a set of call sequences as output. Note that the method trace may comprise a call (or a series of calls) to a single method. In this case, MARBLES outputs an empty sequences set since a sequence must contain at least two distinct items. However, we record the single call as sequence, since it can be useful to tackle API violations like *open-close principle* [89]. The s_i is a set of unique call sequences $\mathcal{X}$. The final set of the call sequences $S_{\mathcal{T}_m}$ (Equation 4.1) for the method m() in test case $\mathcal{T}$ is the union of the set of call sequences s_i.

$$S_{\mathcal{T}_m} = \bigcup_{i=1}^{n} s_i \tag{4.1}$$

4.3.3 Calculating the Hit-Spectrum

The call sequences of a method m() in sequenced spectrum analysis are obtained by running the set of failing test cases (denoted as $\mathbb{T}_F$) and the set of passing test cases (denoted as $\mathbb{T}_P$). We obtain a set of call sequences $\mathcal{S}_m$ (Equation 4.2) for each method. The call sequences set $\mathcal{S}_m$ is the union of (i) the call sequences of a method resulting from the failing test cases ($\mathcal{S}_{\mathcal{T}_m} : \mathcal{T} \in \mathbb{T}_F$) and (ii) the call sequences resulting from the passing test cases ($\mathcal{S}_{\mathcal{T}_m} : \mathcal{T} \in \mathbb{T}_P$).

$$\mathcal{S}_m = \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_F\} \cup \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_P\} \tag{4.2}$$

The set $\mathcal{S}_m$ (Equation 4.2) is used to construct the test coverage matrix for a method. The hit spectrum is then calculated for each call sequence $\mathcal{X}$ in set $\mathcal{S}_m$ from the test coverage matrix.

4.3.4 Ranking Methods

To produce a ranked list of methods, first each call sequence gets a suspiciousness score. Then, a method gets the suspiciousness which is the maximum suspiciousness of its constituting call sequences. The details for these steps follow.

Suspiciousness per call pattern. Based on the hit-spectrum calculated from the test coverage matrix for each method, each call sequence $\mathcal{X} \in S_m$ (Equation 4.2) gets a suspiciousness score $Susp(\mathcal{X})$ calculated by using a fault locator.

Suspiciousness per method. Each method m() gets a suspiciousness $Susp(m)$ which is the suspiciousness of the call sequence $\mathcal{X}$ with the highest suspiciousness (Equation 4.3). We choose the maximum (instead of average) for the suspiciousness score because the technique is looking for exceptional sequences: one unique and highly suspicious sequence is more important than several unsuspicious ones. The methods with an empty call sequence get suspiciousness 0.

$$Susp(m) = \max\Big(\{Susp(\mathcal{X}) \mid \mathcal{X} \in S_m\}\Big) \tag{4.3}$$

Ranked list. Finally, a ranked list of all executed methods is produced by sorting the methods on their suspiciousness $Susp(m)$ such that methods with the highest suspiciousness appear at the top.

4.4 EVALUATION

In this section, we provide the details of empirical evaluation on how far the two variants (raw spectrum analysis and sequenced spectrum analysis) can improve the fault localisation.

4.4.1 Dataset

For this empirical evaluation, we use real defects which have been collected by Just et. al. into a database called Defects4J[2] (a **d**atabase of **e**xisting **f**aults to **e**nable **c**ontrolled **t**esting **s**tudies for **J**ava programs) [59]. Defects4J version 1.1.0 contains defects from 6 open source java projects: Apache Commons **Math**, Apache Commons **Lang**, Joda-**Time**,

[2] http://defects4j.org

Table 4.1: Descriptive Statistics for the Projects Used in Our Experiments

Project	Number of Bugs	Source KLoC	Test KLoC	Number of Tests
Math	106	85	19	3,602
Lang	65	22	6	2,245
Time	27	28	53	4,130
Chart	26	96	50	2,205
Closure	133	90	83	7,927

JFree**Chart**, Google **Closure** Compiler, and **Mockito**. In our study, our tracing system could not be used with Mockito, hence this project was excluded from our study. The descriptive statistics of 5 projects are reported in Table 4.1.

The database contains meta info about each defect including the source classes modified to fix the defect, the test cases that expose the defect, and the test cases that trigger at least one of the modified classes. In this evaluation, we use 346 defects which are located inside methods or constructors.

4.4.2 Evaluation Metrics

Fault localisation heuristics produce a ranked list of elements under test; in the ideal case the faulty unit appears on top of the list. Several ways to evaluate such rankings have been used in the past, including relative measures in relation to project size, such as the percentage of units that need or need not be inspected to pinpoint the defect [60]. However, absolute measures are currently deemed better for comparison purposes [60, 71]. The most commonly adopted metrics are *wasted effort*, *acc@n*, and *mean average precision* [8, 20, 60, 70]. Consequently, we will use these metrics for our comparisons.

To deal with defects spread over multiple locations, we evaluate from the perspective of a *best-case debugging* scenario as argued by Pearson et. al. [85]. In such a scenario identifying one of the possible locations is good to understand and consequently repair the defect. Indeed, once the first faulty element is located it will help developers to find the remaining ones [8].

Mean Wasted Effort (MWE) — Smaller is better. The *mean wasted effort* is the simply the mean of the *wasted effort* in all ranked lists. The *wasted effort* is an absolute measure which

indicates the number of non-faulty methods to inspect in vain before reaching the first faulty method. It is computed as follows:

$$\text{wasted effort} = m + \frac{n}{2} \tag{4.4}$$

Where *m* is the number of non-faulty methods ranked strictly higher than the faulty method; and *n* is the number of non-faulty methods with equal rank in the ranked list to the faulty method. This deals with ties in the ranked list.

acc@n — Higher is better. This is the count of all the faults successfully localised in top-n positions in the ranked list. Inspired by Le et al. [20], we also choose n ∈ {1, 3, 5}, thus effectively creating three variants of the *acc@n* namely *acc@1*, *acc@3*, and *acc@5*. It is not uncommon for two methods in a ranked list sharing the same suspiciousness scores. Hence, while computing the *acc@n*, we break the ties randomly.

Mean Average Precision (MAP) — Higher is better. The *mean average precision* has traditionally been used in information retrieval to evaluate the ranked lists and is also adopted for studying fault localisation. It takes all faulty elements into account and emphasises recall over precision. Thus, it is suitable in scenarios where developers search deep in the ranked list to find more relevant faulty elements [8]. The *mean average precision* is the mean of *average precision* in all ranked lists. The *average precision* in a single ranked list is calculated as following:

$$\text{average precision} = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{number\ of\ faulty\ methods} \tag{4.5}$$

Where: *i* indicates the position of a method in the ranked list; *M* is size of the ranked list (number of methods ranked); *pos*(*i*) is a boolean indicating whether or not the method at i^{th} position in the ranked list is faulty; *P*(*i*) is the precision at i^{th} position in the ranked list, computed as $P(i) = \frac{\#\ faulty\ methods\ in\ top\ i}{i}$.

Our use of several metrics together evaluates fault localisation in several contexts. *Wasted effort* does not normalise the rank of faulty methods with respect to total number of methods in the program. Thus, it is inline with recommendations of Parnin et. al. [71] that for the fault localisation to be useful for developers the aim should be to improve absolute rank rather than percentage rank. In their study, they found that developers switched to other means of debugging when they did not find faulty statements within the first few top positions in the ranked list. The same concerns are also addressed by *acc@n*. However, when developers want to search deep in the ranked list to find more relevant faulty methods, *mean average precision* is suitable in this context [8].

4.4.3 Experimental Protocol

To compare the two variants, we use faulty version of each project. Then, we run each spectrum based fault localisation for all *relevant* test cases, i.e. all test classes which trigger at least one of the source classes modified to fix the fault as recorded in the Defects4J dataset. As such, we obtain ranked lists for each of the 346 defects in the dataset altogether and also organised by project (see Table 4.1). We first compare the 47 fault locators within each spectrum analysis before going into a more detailed analysis. We use five different metrics for this comparison: Mean Average Precision (MAP), Mean Wasted Effort (MWE) and acc@1, acc@3, and acc@5.

Best Performing Fault Locator. While comparing raw spectrum analysis against sequenced spectrum analysis we use the best performing fault locator for each case. To identify the best performing fault locator, we first rank all 47 fault locators on each of the five evaluation metrics and then compute the mean of the ranks. Thus, the fault locator with the lowest mean rank— performing best in all evaluation metrics— is selected as the best performing one.

Significance Tests. We perform statistical tests of significance for raw spectrum analysis and sequenced spectrum analysis on the evaluation metrics *wasted effort* (*WE*) and *average precision* (*AP*). Since we have a matched pair design and we compare whether one variant is better than its counterpart, we choose the Wilcoxon signed rank test and run as paired one-tailed test. We favour the non-parametric Wilcoxon signed rank test over parametric t-test owing to small sample sizes and non-normal distribution of scores for both *wasted effort* as well as *average precision*. As is common practice in software engineering research, we set the significance level α of 0.05 (there is 5% risk of concluding that the two distributions are different when in fact they are not).

Research Questions. In this evaluation, we address following research questions.

RQ1. *What is the baseline performance of raw spectrum analysis?*

Motivation. We establish the best performing fault locator and obtain the rankings for the 346 faults in the Defects4J. This sets the baseline against which we compare.

RQ2. *How much can sequenced spectrum analysis improve upon raw spectrum analysis?*

Motivation. We establish the best performing fault locator for sequenced spectrum analysis and obtain the rankings for the same 346 faults. We compare these rankings against the baseline obtained in **RQ1.**.

RQ3. *Are there project specific differences between the rankings?*

Motivation. Inspired by the work of Zeller et. al. [90], we investigate whether the results obtained for the whole Defects4J data set apply to the various projects within that dataset. This is to assess the robustness of our findings.

RQ4. *Is sequenced spectrum analysis efficient compared to raw spectrum analysis?*

Motivation. Here, we evaluate the running time of sequenced spectrum analysis in comparison with raw spectrum analysis to note its practical applicability.

Additional material that was excluded from the original paper due to space constraints.

4.5 RESULTS

In this section, we discuss the answers to three research questions introduced in Section 4.4.3.

RQ1. *What is the baseline performance of raw spectrum analysis?*

To answer this question, we apply raw spectrum analysis with 47 known fault locators on all defects together, thus aggregating the results over all projects in the dataset. This allows for a sufficiently rigorous analysis of the current state of the art and as such establishes the baseline performance of raw spectrum analysis. As mentioned in the protocol, we rank the fault locators from the top with the best performance to the bottom with the worst.

Table 4.2 lists the fault locators along with their scores for five evaluation metrics sorted on their rank (rightmost column entitled R). Fault locators with the same rank are highlighted in the same background colour. We observe in the table that GP13 and Naish2 have good scores for acc@1, acc@3, acc@5, and a better mean average precision than M2 and Goodman. M2 and Goodman, on the other hand, are slightly better in terms of mean wasted effort. Studying the performance on each of the evaluation metrics, the value of 63 for acc@1 tells us that for 63 out of 346 (18%) raw spectrum analysis has an exact hit: the method containing the fault is the first one in the ranking. Similarly, acc@3 (120 out of 346 is 35%) and acc@5 (142 out of 346 is 41%) demonstrates that the fault locators perform reasonably well in many cases. The *mean wasted effort* (MWE), however, reveals that for many cases the fault localisation is unsatisfactory: a MWE of 96 implies that on average 96 methods need to be inspected before one arrives at the correct location. This suggests a long tail distribution, where for many cases the first faulty method is ranked quite low. The value for *mean average precision* (MAP) reinforces the problem: a low value of 0.27 implies that the relative location of other faulty methods (in cases where fault expands to multiple methods) is also quite low.

Table 4.2: Establish the baseline performance for raw spectrum analysis over the 346 defects in the dataset.

MAP = Mean Average Precision, MWE = Mean Wasted Effort.

Fault locator	acc@1	acc@3	acc@5	MAP	MWE	Rank
GP13 [18]	63	120	142	0.2780349	96.73	1
Naish2 [17]	63	120	142	0.2776756	96.64	1
M2 [17]	62	118	141	0.2753030	96.32	2
Goodman [17]	61	116	138	0.2695181	16.68	3
Ample2 [17]	64	120	140	0.2764775	101.24	3
T* [60]	62	119	139	0.2744910	96.37	4
Zoltar [91]	61	118	138	0.2735461	96.14	5
Kulczynski2 [17]	61	116	137	0.2718006	96.56	6
Ochiai [67]	61	116	138	0.2693963	98.02	7
Jaccard [45]	61	116	138	0.2695181	104.20	8
Dice [17]	61	116	138	0.2695181	104.20	8
Kulczynski1 [17]	61	116	138	0.2695181	104.20	8
Anderberg [17]	61	116	138	0.2695181	104.20	8
Sørensen -Dice [17]	61	116	138	0.2695181	104.20	8
GP19 [18]	62	118	139	0.2729868	125.20	9
Arithmetic Mean [17]	61	118	138	0.2677694	102.93	9
Cohen [17]	61	118	138	0.2678460	105.66	10
Harmonic Mean [17]	60	115	135	0.2637967	82.19	10
Geometric Mean [17]	60	115	135	0.2620774	83.72	11
Fleiss [17]	58	107	123	0.2471480	36.29	12
Scott [17]	57	107	125	0.2446414	37.93	13
CBIInc [17]	60	118	135	0.2669550	107.01	13
Barinel [92]	60	118	135	0.2669550	107.11	14
Tarantula [11]	60	118	135	0.2668479	107.03	14
CBISqrt [17]	62	115	136	0.2657316	125.09	15
Rogot2 [17]	60	115	136	0.2642806	139.41	16
Ochiai2 [17]	60	115	136	0.2634117	130.00	17
Wong3′ [17]	42	72	86	0.1785421	22.64	18
Wong3 [93]	42	72	86	0.1785421	22.64	18
Wong2 [93]	41	66	77	0.1582824	16.04	19
Rogot1 [17]	57	107	125	0.2446351	338.34	20
Ample [67]	54	93	116	0.2297198	214.02	21
CBILog [17]	15	28	36	0.0757946	33.22	22
Overlap [17]	41	77	100	0.1857891	178.36	23
Russell & Rao [17]	38	73	97	0.1802516	177.87	24
Wong1 [93]	38	73	97	0.1802516	177.87	24
Binary [17]	37	69	92	0.1714134	188.00	25
Hamann [17]	41	66	77	0.1582824	367.43	26
Hamming etc. [17]	41	66	77	0.1582824	437.74	27
Rogers & Tanimoto [17]	41	66	77	0.1582824	437.74	27
GP02 [18]	23	48	63	0.1268758	277.10	27
M1 [17]	41	66	77	0.1582824	437.74	27
Simple Matching [17]	41	66	77	0.1582824	437.74	27
Sokal [17]	41	66	77	0.1582824	437.74	27
GP03 [18]	12	23	32	0.0656960	342.97	28
Euclid [17]	7	14	27	0.0521883	428.46	29
Naish1 [17]	1	3	7	0.0200899	419.36	30

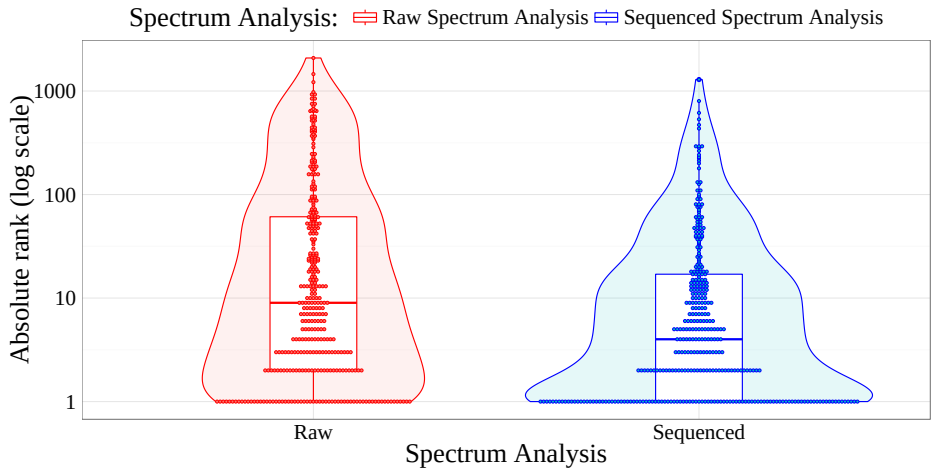


Figure 4.1: Comparison of the distribution of absolute rankings for faulty methods.

As baseline performance, we deduce that 18% of faults correspond with an exact hit (acc@1) while for many faults the heuristic performs reasonably well (acc@3 for 35% of the faults; acc@5 for 41% of the faults). However, the mean wasted effort is 96.73 which implies that for many cases the fault localisation is unsatisfactory and suggests a long tail distribution.

RQ2. *How much can sequenced spectrum analysis improve upon raw spectrum analysis?*

To answer this question, we apply sequenced spectrum analysis over the same dataset using the same protocol. Thus, we use the fault locators on all 346 defects and rank them to identify the best performing one. This allows for a fair comparison between raw spectrum analysis and sequenced spectrum analysis in the sense that we choose the optimal configuration for both of them. Table 4.3 lists the fault locators along with their scores for five evaluation metrics sorted according to their rank; highlighting fault locators with the same rank in the same background colour.

Here, we see that Ample2 is the best performing fault locator with Fleiss, T*, M2, and Arithmetic Mean as close seconds. However, the scores of Ample2 with sequenced spectrum analysis are better for all evaluation metrics compared to the raw spectrum analysis. The value for acc@1 is 103, thus for 103 out of 346 (30%) faults sequenced spectrum analysis has an exact hit — an absolute 12% improvement (30% vs 18%). Similar improvements can be seen for acc@3 (159 out of 346 is 46% compared to 35%) and acc@5 (191 out of 346 is 55% compared to 41%). Also, the mean wasted effort has reduced from 96.73 to 25.88, thus on average the fault is now located on the 25th position in the ranking. The mean

Table 4.3: Performance improvement for sequenced spectrum analysis over the 346 defects in the dataset.

MAP = Mean Average Precision, MWE = Mean Wasted Effort.

Fault locator	acc@1	acc@3	acc@5	MAP	MWE	Rank
Ample2	103	159	191	0.3925699	25.88	1
Fleiss	103	157	191	0.3874628	16.01	2
T*	102	160	190	0.3936098	31.18	3
M2	102	160	189	0.3933028	31.10	4
Arithmetic Mean	103	157	189	0.3875244	26.67	5
Goodman	101	157	190	0.3854730	9.83	6
Naish2	101	159	187	0.3918473	29.12	7
Geometric Mean	102	156	188	0.3866599	24.31	8
Kulczynski2	101	157	189	0.3893806	30.53	9
Ochiai	101	158	191	0.3883824	31.42	10
GP19	103	159	190	0.3929647	34.38	10
GP13	101	159	188	0.3927791	33.46	11
Harmonic Mean	102	154	186	0.3859800	23.83	12
Scott	101	156	189	0.3821700	16.73	12
Cohen	101	157	189	0.3851854	26.74	13
Jaccard	101	157	190	0.3866061	32.17	14
Dice	101	157	190	0.3866061	32.17	14
Anderberg	101	157	190	0.3866061	32.17	14
Sørensen -Dice	101	157	190	0.3866061	32.17	14
Ochiai2	101	157	192	0.3863531	35.48	15
Kulczynski1	101	154	187	0.3848006	32.52	16
Rogot2	102	154	186	0.3860920	37.45	17
Wong3′	89	139	167	0.3391367	12.15	18
CBIInc	96	147	182	0.3738493	28.32	18
Wong3	89	139	167	0.3391367	12.15	18
CBISqrt	99	154	189	0.3817832	34.71	19
Rogot1	101	156	189	0.3820858	61.86	20
Wong2	87	132	157	0.3222868	9.79	21
Zoltar	96	153	183	0.3739687	32.60	22
Tarantula	96	147	182	0.3750226	34.01	23
Barinel	96	147	182	0.3750091	34.03	24
Ample	94	146	176	0.3612457	44.78	25
CBILog	38	63	82	0.1650905	8.73	26
Hamming etc.	87	132	157	0.3222977	91.93	27
Rogers & Tanimoto	87	132	157	0.3222977	91.93	27
Hamann	87	132	157	0.3222868	83.37	27
Simple Matching	87	132	157	0.3222977	91.93	27
Sokal	87	132	157	0.3222977	91.93	27
M1	87	130	155	0.3211702	92.57	28
Russell & Rao	51	96	118	0.2328541	113.18	29
Wong1	51	96	118	0.2328541	113.18	29
Binary	45	88	109	0.2098345	125.15	30
Overlap	43	80	104	0.2018613	114.95	31
GP02	31	69	88	0.1702866	176.30	32
GP03	31	69	89	0.1632317	175.70	32
Naish1	16	28	45	0.0900345	142.08	33
Euclid	8	18	30	0.0681149	268.55	34

average precision has risen from 0.27 to 0.39 implying that besides the location of the first faulty methods, the location of other faulty methods has also improved— the faulty methods are ranked higher in the list .

This suggests that the distribution of the rankings is better for sequenced spectrum analysis, which is confirmed in Figure 4.1. In these violin plots we juxtapose the distributions of absolute ranks (i.e. the location of the first faulty method) for both variants. There are some interesting observations in these plots. First, the density curve in the plot for sequenced spectrum analysis is wide for lower ranks and quickly narrows for higher ranks, indicating that most of the faulty methods are located on top of the ranked list. The density curve for raw spectrum analysis on the other hand narrows slowly indicating that many faulty methods are also located deeper in the ranked list — the ranks are spread. Second, the median in the box plot for sequenced spectrum analysis shows that for 50% of the faults the faulty methods are located within location 4 in the ranked lists, whereas for raw spectrum analysis this median is at 9 implying the faulty methods are located more deeply. Finally, the third quartile for sequenced spectrum analysis is nearly same as the median for raw spectrum analysis— the highest location where half of the faults are ranked in raw spectrum analysis, with sequenced spectrum analysis about 75% of the faults are located at the same location.

Significance tests for sequenced spectrum analysis versus raw spectrum analysis on both average precision and wasted effort metrics have p-values **< 2.2e-16**, show that sequenced spectrum analysis is not only better, but that it is *significantly* better in the statistics sense of the word.

> *When compared to raw spectrum analysis, sequenced spectrum analysis gains 12% improvement for exact hit (acc@1) and reduces the average wasted effort from 96.73 to 25.88. The distribution of the fault locations is better which results in statistically significant improvements.*

RQ3. *Are there project specific differences between the rankings?*

While answering the previous questions, we generalised the comparison on all the defects together irrespective of the project. However, as noted by Zeller et. al. there are project-specific variations that might provide valuable insights [90]. Therefore, we compare the two spectrum analyses on defects for each individual project. As done in previous subsections, we first select the best performing fault locator for each project and for each variant. Due to space limitations, we omit the results for the selection of the best performing fault locator and immediately move towards the actual comparison in Table 4.4. This table lists the project specific scores for the five evaluation metrics for

Table 4.4: Project specific comparison of sequenced spectrum analysis (SS) versus raw spectrum analysis (RS).

SA = Spectrum Analysis (SS vs RS), MAP = Mean Average Precision, MWE = Mean Wasted Effort.

Project	SA	Fault Locator.	acc@1	acc@3	acc@5	MAP	MWE
Closure	SS	Fleiss	17	37	47	0.2236320	30.61
	RS	GP13	7	15	20	0.1085065	222.89
Math	SS	Goodman	33	54	69	0.4458375	4.79
	RS	Goodman	21	45	51	0.3349293	7.92
Lang	SS	Geometric Mean	41	50	52	0.7294623	1.167
	RS	GP13	21	43	49	0.5238196	3.78
Time	SS	Ample2	6	10	15	0.2938999	16.15
	RS	GP13	5	8	9	0.2201962	39.38
Chart	SS	CBISqrt	9	14	16	0.4632694	13.2
	RS	Tarantula	10	15	16	0.4986104	27.16

the best performing fault locator for that project. Table 4.5 also lists the project specific comparison of p-values for both average precision and wasted effort.

The first interesting observation to make concerns the best performing fault locators: they vary a lot across projects. Naish2 (the best performing fault locator for raw spectrum analysis when applied on all projects together) is never the best performing project-specific fault locator. And Ample2 (the best performing fault locator for sequenced spectrum analysis when applied on all projects together) only appears as the best for the Time project. Thus, a new set of best performing fault locators has emerged for both raw spectrum analysis and sequenced spectrum analysis on project by project basis, illustrating that great care is needed when configuring such tools.

The second interesting observation in Table 4.4 is that there is positive change for the values of acc@1 for both spectrum analyses. With a new set of best performing fault locators, overall exact hit (acc@1) for sequenced spectrum analysis has now increased from 103 to 106, while for raw spectrum analysis it has increased from 63 to 64.

Next, we see that —with the exception of project Chart— sequenced spectrum analysis outperforms raw spectrum analysis. Moreover, the p-values in Table 4.5 confirm that

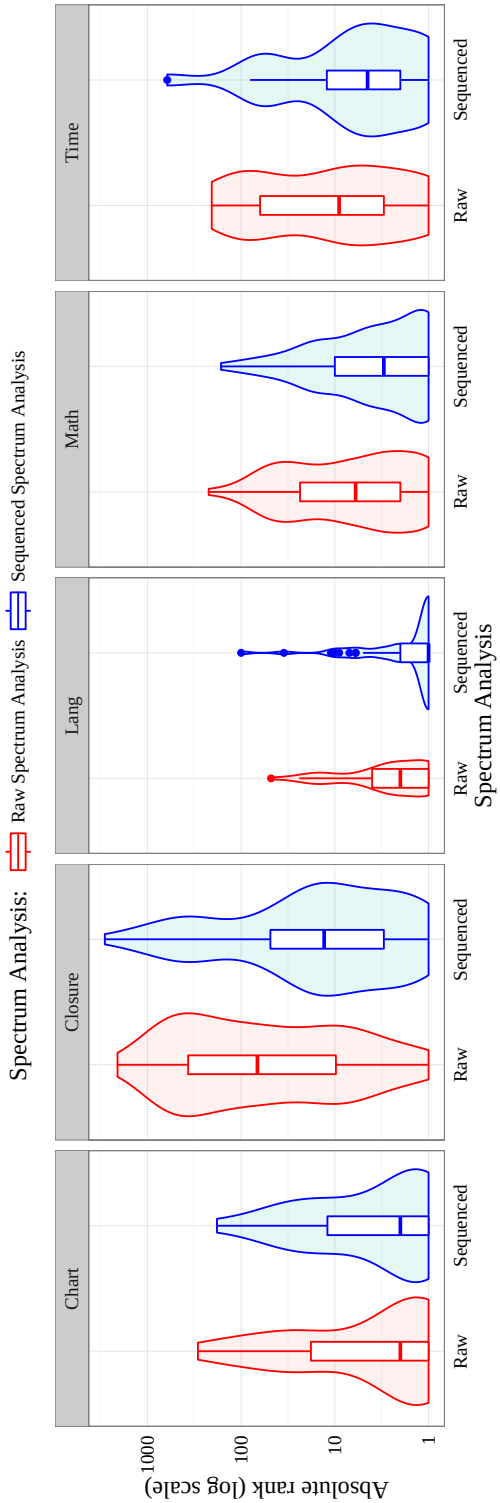


Figure 4.2: Distributions of absolute ranks of faulty methods for both spectrum analyses for each project.

Table 4.5: Significance tests for sequenced spectrum analysis vs. raw spectrum analysis.

Project	Comparison	p-value (Average Precision)	p-value (Wasted Effort)
Closure	SS > RS	**7.849e-10**	**< 2.2e-16**
Math	SS > RS	**6.983e-05**	**9.915e-07**
Lang	SS > RS	**2.87e-05**	**2.096e-05**
Time	SS > RS	**0.01442**	**0.0009032**
Chart	SS > RS	—	0.2764
	RS > SS	0.3638	—

sequenced spectrum analysis is statistically significantly better for these four projects, however the p-value for average precision in project Time is insignificant. A deeper analysis of the project Chart reveals that sequenced spectrum analysis is better for mean wasted effort while raw spectrum analysis is better for mean average precision and acc@1. However, as seen in Table 4.5 the better score for mean wasted effort with sequenced spectrum analysis is statistically significant while the better score for mean average precision with raw spectrum analysis is statistically insignificant.

We again turn to violin plots to explore these differences in more detail. Figure 4.2 provides distributions of absolute ranks of first faulty method in the ranked lists for both spectrum analyses for each project. We readily observe that shapes of the two distributions for the project Chart are nearly the same, suggesting that sequenced spectrum analysis slightly improves upon raw spectrum analysis. However, sequenced spectrum analysis improves upon raw spectrum analysis for the remaining four projects— with significant improvement for the project Lang. Yet, we observe that the distribution of absolute ranks with sequenced spectrum analysis for project Time has some outliers.

On project by project basis, sequenced spectrum analysis performs better than raw spectrum analysis for four projects. For the fifth project the results are, for practical purposes, the same. Moreover, the best performing fault locator varies a lot across the projects.

RQ4. *Is sequenced spectrum analysis efficient compared to raw spectrum analysis?*

All the measurements are performed on a Mac machine (2.5 GHz Intel Core i7, 16 GB 1600 MHz DDR3) running MacOSX (10.11.6) using the bash internal

Table 4.6: Summary of time taken by each spectrum analysis for all projects.

Spectrum Analysis	Project	Time			
		Tracing	Sequence Generation	Ranking	Total
sequenced spectrum analysis	Closure	03:18:58	69:40:54	03:19:23	16:19:15
	Math	01:09:09	01:20:35	00:01:51	02:31:35
	Lang	00:08:30	00:20:37	00:00:48	00:29:55
	Time	00:04:57	00:04:57	00:01:05	00:10:59
	Chart	00:06:07	00:01:39	00:00:28	00:08:14
raw spectrum analysis	Closure	01:09:22	—	00:02:39	01:12:01
	Math	01:07:29	—	00:01:09	01:08:38
	Lang	00:10:13	—	00:00:35	00:10:48
	Time	00:03:24	—	00:00:19	00:03:43
	Chart	00:04:04	—	00:00:16	00:04:20

Additional material that was excluded from the original paper due to space constraints.

variable $SECONDS (which indicates the number of seconds the script has been running) for a single run of the heuristic. As we run the tests sequentially, such time measurements are crude, thus should only be seen as an initial indicator for the relative time during the different steps.

Apart from the time to collect the coverage and produce the ranked lists, sequenced spectrum analysis incurs an overhead to mine sequences using the *MARBLES algorithm* [88]. Thus, there is an additional time overhead of the *MARBLES* during sequenced spectrum analysis. Table 4.6 provides the timing information with both techniques. We observe that the techniques are practically applicable for smaller projects, with sequenced spectrum analysis relatively slower owing to extra overhead. However, for large project Closure, we see the sequenced spectrum analysis running the risk of being impractical.

Additional material that was excluded from the original paper due to space constraints.

4.6 RELATED WORK

The Tarantula tool provided the foundation for research on spectrum based fault localisation [11]. Afterwards, several researchers made attempts to increase the effectiveness of spectrum based fault localisation, including work on (a) finding the optimal *fault locators*, (b) changing the *spectrum analysis*, (c) using state-of-the-art information retrieval techniques to *learn to rank*, and (d) *test-suite reduction and diagnosability*.

Fault locators. Abreu et al. introduced Ochiai, used in the molecular biology domain, into spectrum based fault localisation and demonstrated better performance [15, 67]. Steimann et. al. defined and evaluated T* (a variant of Tarantula) and there as well demonstrated better performance [60]. Lucia et al. applied 20 well-known association measures from data mining on fault localisation and concluded that 10 out of 20 association measures were comparable to Tarantula and Ochiai [16]. Naish et al. proposed a couple of fault locators through a theoretical model and established that they performed better than existing ones [17]. Later studies confirmed that one (Naish2) is among the best performing fault locators [20, 85], which is corroborated in this comparison. Yoo evolved an entirely different set of fault locators (GP01 ...GP30) that performed better than existing ones [18]. Work by B. Le et al. finds that GP13 and GP19 perform better [20].

Hit-Spectra. Yilmaz et al. proposed time-spectrum as a variation for spectrum analysis [19]. Instead of coverage of methods, time-spectrum uses traces of method execution times collected from both passing and failing tests. The potential causes of faults are identified as deviations of failing tests from behaviour models created from time spectra collected in passing test runs. Dallmeier et al. extracted sequence of method calls by sliding a window of fixed size over execution traces of classes to identify faulty classes [22]. Likewise, Laghari et al. pinpoint faulty classes but adopting itemset mining [30]. However, in our approach, we use sequence mining to mine the sequences from call traces of methods to locate faulty methods and not classes.

Learning to rank. Xuan and Monperrus proposed MULTRIC, a learning-based approach which combines multiple ranking metrics to learn and then rank [70]. They demonstrated on seeded faults that MULTRIC improved upon existing fault locators. Similarly, B. Le et al. [20] propose *Savant*, a learning to rank approach which exploited inferred likely method invariants mined from passing and failing test cases. They find that *Savant* is more effective than state of the art on real faults.

Specification mining. Runtime traces have been used to learn API specifications such as legal method call sequences. These specifications are used for purposes including documentation, learning the APIs, and also for bug detection. OCD learns and enforces temporal specifications over method call sequence [94]. The algorithm uses a predefined template which specifies a sequence of only two method calls and operates over a finite

window on the trace. The tool is reported to have detected anomalies as violations of inferred sequences in Eclipse and Ant, though the anomalies did not result in program crashes. Pradel and Gross infer specification for Java standard library from method traces. They use method calls as object collaborations to infer API specifications as finite state machines which model the legal method call sequence [41]. JMiner traces Java programs to generate parametric specifications. The specifications produced with JMiner are reported to have detected a few bugs in open source Java programs [95]. Similarly, we mine call sequences for methods from both passing and failing tests and statistically compare these sequences to pinpoint faulty methods.

4.7 THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [80, 81]), we organise them into four categories.

Construct validity. In this research, we compare sequenced spectrum analysis against raw spectrum analysis. To reduce the risk on construct validity, we evaluated with five different metrics assessing different perspectives on what is deemed better. The use of an absolute metric (wasted effort) and also acc@n alleviates concerns on relative measures [71]. While the evaluation of fault localisation on mean average precision has implication for developers who search deep in the ranked list to find more relevant faulty methods and for automated fault repair techniques.

Internal validity. When selecting the best fault locator which performs better in all five evaluation metrics, we use simplified method of first ranking the fault locators on individual metrics, then computing the mean of their ranks, and finally rank them on their mean rank. This may not be the best solution but we ensured that it was better than simple aggregation method of summation of metric scores.

External validity. We use several evaluation metrics together which implies a stringent comparison. Evaluation on a single metric alone may result in a different interpretations. A notable example in this chapter is comparison on project Chart (see Table 4.4). If only evaluated on *Mean Wasted Effort*, the sequenced spectrum analysis is better than raw spectrum analysis. However, when comparing on several metrics together the result is different. This observation signals that the evaluation metric used to evaluate the fault localisation has an effect on its accuracy. Thus, it is also unwise to generalise the findings, but instead metric-specific insights should be explored.

Reliability. All the tools involved in this case study (i.e. creating the traces, calculating the ranked lists etc.) have been implemented and tested for three years by the first

author. Moreover, for sequenced spectrum analysis we used the MARBLES [88] algorithm which has been already tested by the creators of MARBLES. However, lurking faults in any of the tools may affect the precise rankings.

4.8 CONCLUSION

In this chapter, we presented sequenced spectrum analysis— a class of spectrum based fault localisation which modifies the hit-spectrum by incorporating series of method calls mined from the execution traces. To compare sequenced spectrum analysis against the state of the art, we created a suite of fault localisation heuristics with 47 known fault locators and evaluated them to establish a baseline with the best performing one. Then, we compared sequenced spectrum analysis against raw spectrum analysis and conclude that sequenced spectrum analysis is better than raw spectrum analysis, regardless of whether we evaluate for the whole dataset or whether we evaluate on a project specific basis.

During this comparison, we also learned that the best performing fault locator varies quite a lot depending on the project, the variant of spectrum analyses (raw spectrum analysis and sequenced spectrum analysis), and even the experimental setting (all defects, defects per project basis). Finally, the evaluation metrics do also play a role: we observed some cases where the best performing fault locator varies with the evaluation metric used (Mean Average Precision, Mean Wasted Effort, acc@n).

These observations have few important consequences for future research in fault localisation. First, choosing the best fault locator is highly context specific, depending on both the project and the experimental set-up, therefore these factors need to be considered before generalising the conclusions. Second, the evaluation metric used to assess the performance of fault localisation does also matter when drawing the conclusions.

4.9 ACKNOWLEDGMENTS

Thanks to Boris Cule for helping with MARBLES algorithm. This work is sponsored by (a) the Higher Education Commission of Pakistan under a project titled “Strengthening of University of Sindh (Faculty Development Program)"; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

Chapter 5

Spectrum Based Fault Localisation: What about Component Tests ?

ABSTRACT

Agile testers distinguish between unit tests and component tests as a way to automate the bulk of the developer tests. Research on spectrum based fault localisation largely ignores this distinction, evaluating the effectiveness of these techniques irrespective of whether the fault is exposed by unit tests—where the search space is constrained to the unit under test— or by component tests—where the search space expands to all objects involved in the test. In this chapter, we evaluate sixteen spectrum based fault localisation techniques and demonstrate that the performance depends a lot on the presence of faults exposed by unit tests and component tests in the dataset. Therefore, we urge researchers in fault localisation to distinguish between easy and difficult faults in future evaluations.

5.1 INTRODUCTION

Software testing is the activity of executing a program with the intent of finding a defect. A software test brings the implementation under test in a given state, then administers a sequence of stimuli and subsequently verifies whether the resulting state corresponds with the expectations. Once a software test exposes a defect, a software engineer still has to search for its root cause —the *fault*— and fix it accordingly. To minimise the search space, testing handbooks distinguish between *unit tests* and *component tests* [78, 96]. A unit test isolates the implementation under test (typically a method or a class) from the rest of the system so that the tester can be confident that the fault is located within the unit. A component test, on the other hand, exercises the interactions between objects; when a

component test exposes a defect, the fault should be in the code that manipulates the interface. Unfortunately, one cannot entirely rule out the code in the constituting objects (even with the presence of stubs and mock objects), thus the search space for locating the fault comprises all the objects involved in the component test.

To help software engineers locate the root cause of a defect, the research community has forwarded *spectrum based fault localisation* techniques [20, 21, 31, 85, 97]. These produce a ranked list of program elements, indicating the likelihood of a program element being at fault. They do so by analysing the program traces generated by failing and passing tests.

Given that unit tests and component tests represent different strategies to pinpoint the location of a fault, one would expect that the research on fault localisation also makes this distinction. However, none of the currently published evaluations do so: all of them rely on datasets such as the Siemens set [72]; the Software-Artifact Infrastructure Repository (SIR) [46], iBugs [74], and the most recent Defects4J [59]. None of these datasets distinguish between unit tests and component tests, hence it is currently unknown how fault localisation heuristics deal with the more challenging faults involving a larger search space.

To illustrate the differences, we showcase two examples from Defects4J. An easy case for fault localisation is fault 3 in project Math. The unit test testLinearCombinationWithSingleElementArray in test class MathArraysTest calls only a single method (the one containing the fault) —linearCombination(double[ ], double[ ]) in class MathArrays. The test fails because of an ArrayIndexOutOfBoundsException, which is immediately visible in the stack trace and readily points to the location of the fault. In such cases, *any* fault localisation technique —even the most naive one— should have an accuracy of 100%. In contrast, one of the most difficult faults to locate is fault 74 in project Math. The test case polynomial in class AdamsMoultonIntegratorTest exposes a fault in method integrate(FirstOrderDifferentialEquations, double, double[ ], double, double[ ]) within class EmbeddedRungeKuttaIntegrator. The test fails because one assertion fails: the returned value does not match the expected value because of some erroneous state manipulation earlier in the implementation under test. The faulty method does not appear in the stack trace, so one needs to search through all the 264 project methods indirectly called by the test case. Such needle-in-a-haystack cases are real challenges for fault localisation techniques because the search space of potential fault locations is so large.

In this chapter, we evaluate sixteen spectrum based fault localisation techniques to see how they fare on faults exposed by unit tests and component tests. As such, we make the following contributions.

1. We refine the Defects4J dataset [59]. We separate faults into two categories: the

ones exposed by unit tests (where the search space is rather small) and the ones exposed by component tests (where the search space is larger).
2. We assess the size of the search space for both unit tests and component tests, showing that there is indeed a significant difference.
3. We construct a comprehensive suite of spectrum based fault localisation techniques resulting in sixteen different techniques.
4. We evaluate the performance of these sixteen techniques and demonstrate that the performance depends a lot on the presence of unit test and component test related faults in the dataset.

In the remainder of this chapter, we list the known variants in spectrum based fault localisation used during the evaluation in Section 5.2, and then describe the set-up of the evaluation in Section 5.3, which naturally leads to Section 5.4 reporting the results. After a discussion on the threats to validity in Section 5.5 and the related work in Section 5.6, we come to a conclusion in Section 5.7.

5.2 FAULT LOCALISATION TECHNIQUES

Spectrum based fault localisation is quite an effective class of techniques as reported in several papers [8, 20, 21, 31, 67, 85, 98]. To understand how these techniques work, there are four crucial elements to consider: (1) the test coverage matrix; (2) the granularity; (3) the hit-spectrum; and (4) the fault locator. We explain each of them below.

Test Coverage Matrix. All spectrum based fault localisation techniques collect coverage information of the elements under test in a *test coverage matrix*. This is a matrix, where the rows correspond to elements under test and the columns represent the test cases. Each cell in the matrix marks whether a given element under test is covered by the test case (1) or not (0).

Granularity. The test coverage matrix conceals an important variation point in spectrum based fault localisation: the granularity of the analysis. Different levels of granularity have been studied including *statements* [11, 64, 65, 85], *blocks* [16, 67, 68, 69], *methods* [10, 20, 21, 31, 60, 70, 98], and *classes* [22, 30]. We will focus on method-level granularity in this chapter, for the following reasons. In object-oriented testing a method is the smallest element under test [78]. Also, the stack traces for failing tests are reported at method level. Finally, there is evidence that developers have a slight preference for method-level granularity [87].

Basic Hit-Spectrum. Next, the test coverage matrix is transformed into the *hit-spectrum* (sometimes also called *coverage spectrum*) of a program. The hit-spectrum of an element under test is a tuple of four values (e_f, e_p, n_f, n_p). Where e_f and e_p are the numbers of failing and passing test cases that execute the element under test and n_f and n_p are the

Table 5.1: Popular Fault Locators

Faul Locator	Definition
Barinel [92]	$1 - \frac{e_p}{e_p + e_f}$
D*† [99]	$\frac{{e_f}^*}{e_p + n_f}$
GP13 [18]	$e_f \times \left(1 + \frac{1}{2 \times e_p + e_f}\right)$
GP19 [18]	$e_f \times \sqrt{\lvert e_p - e_f + e_f + n_f - e_p + n_p \rvert}$
Ochiai [67]	$\frac{e_f}{\sqrt{(e_f + n_f)(e_f + e_p)}}$
Op2 [17]	$e_f - \frac{e_p}{e_p + n_p + 1}$
Tarantula [11]	$\frac{\frac{e_f}{e_f + n_f}}{\frac{e_f}{e_f + n_f} + \frac{e_p}{e_p + n_p}}$
T* [60]	$\left(\frac{\frac{e_f}{e_f + n_f}}{\frac{e_f}{e_f + n_f} + \frac{e_p}{e_p + n_p}}\right) \times max\left(\frac{e_f}{e_f + n_f}, \frac{e_p}{e_p + n_p}\right)$

† Inspired from [85], we also use * = 2, which is most thoroughly explored value.

numbers of failing and passing test cases that do *not* execute the element under test [64]. The standard spectrum based fault localisation implementation, GZoltar, produces basic hit-spectrum which is used until recent studies [85].

Extended Hit-Spectrum. Recently new techniques have modified the hit-spectrum via *frequent itemset mining* [31], *inferred invariants* [20], *code and change metrics* [21], and *page rank* [98]. For instance, Laghari et al. proposed to extend the hit-spectrum, leveraging patterns of method calls by means of *frequent itemset mining* [31]. Instead of creating a single test coverage matrix per program, they create a test coverage matrix for each executed method. With the extended hit-spectrum, a row in the test coverage matrix corresponds to a call pattern of the method; the hit-spectrum (e_f, e_p, n_f, n_p) of a call pattern indicates whether or not it is involved in a test case.

Fault Locators. Finally, a spectrum based fault localisation technique assigns a suspiciousness to each element under test by means of a *fault locator*. The *fault locator* essentially

Table 5.2: Descriptive Statistics — Defects4J

Project	$\text{Defects}_{\text{UT}}{}^{\dagger}$	$\text{Defects}_{\text{CT}}{}^{\ddagger}$	$\text{Defects}_{\text{Total}}$
Closure	01 (00.8%)	131 (99.2%)	132
Math	26 (20.2%)	77 (74.8%)	103
Lang	40 (66.7%)	20 (33.3%)	60
Time	01 (03.8%)	25 (96.2%)	26
Chart	05 (20.0%)	20 (80.0%)	25
TOTAL	73 (21.1%)	273 (78.9%)	346

† Defects exposed by unit tests. ‡ Defects exposed by component tests.

translates the *hit-spectrum* into a suspiciousness score. This suspiciousness indicates the likelihood of the element to be at fault. Table 5.1 lists the most popular and best performing fault locators reported in recent studies [20, 31, 60, 85].

For the remainder of this chapter, we construct a suite of spectrum based fault localisation techniques by combining each of the eight best performing fault locators with the basic hit-spectrum and extended hit-spectrum. For the extended hit-spectra, we choose hit-spectrum extended via frequent itemset mining for its simplicity in easy implementation. This results in sixteen different techniques classified into two families. The family of basic hit-spectrum (called Basic) will be marked with an index B (as in $Ochiai_B$), the family of extended hit-spectrum (called Extended) with an index E (as in $Ochiai_E$)

5.3 CASE STUDY SETUP

Given the sixteen different techniques to evaluate, we now establish the criteria used for the assessment. We explain the details about the dataset used for the evaluation (Defects4J); the refinements we made to that data set to distinguish unit tests from component tests; and the evaluation metrics used during the evaluation. We finish with the research questions and the protocol driving the case study.

5.3.1 Refining Defects4J

In our experiments, we use real faults from 5 open source java projects: Apache Commons **Math**, Apache Commons **Lang**, Joda-**Time**, JFree**Chart**, and the Google **Closure** Compiler from the established dataset—Defects4J (version 1.1.0) where the fault is inside a method [59]. Currently, we could not use faults from the Mockito project due to the limitation of our tracer.

The Defects4J dataset does not make any classification of the nature on the tests, hence one cannot deduce whether a given fault is exposed by unit tests or component tests. Therefore, we separate all of the failing tests for a given faulty version in Defects4J into either a unit test or component test. We adopt the definitions by Crispin and Gregory [96]. A unit test isolates the implementation under test (typically a method or a class), whereas a component test exercises the interactions between objects (classes).

Since we have the call traces anyway, we adopt a dynamic analysis heuristic, inspired by the one of Weijers [100]. In this heuristic, we run all the failing test cases and for each test case collect all the project classes called during the test execution, thus excluding all library and framework classes. We use AspectJ to intercept the execution of project methods.

For example, to trace the execution of methods in project Chart the pointcut becomes * org.jfree..*.*(..). The pointcut essentially specifies to intercept the execution of any method in any class in package org.jfree or its subpackages—org.jfree is the top level package for project Chart. Then, we add the class of the intercepted project method in the set of called classes. Since a unit test can also use a "mock" class to mock or simulate the real class, we exclude classes where the name contains variations of the word "mock".

Next, we enumerate all classes in the set of called classes to determine the class under test. We use the metric *name similarity* for textual similarity [101] in the following steps.

- *Package* P: we calculate the similarity between the package name of the test class and the called class (Rationale: the project classes and their corresponding test classes share the same package hierarchy in projects in Defects4J).
- *Class* C: we calculate the similarity between the class name of test class and the called class (Rationale: test classes normally have the same name as the class under test, such as TimeSeriesTests and TimeSeries)
- *Method* M: we calculate the textual similarity between the test method name and each of the methods of the called class and choose the method with the highest textual similarity (Rationale: test methods normally are named by prepending the word "test" to the method name being tested, such as TimeSeriesTests.testCreateCopy3 tests method createCopy in class TimeSeries).

- Finally, the class with highest similarity score, sum(P, C, M), is determined as being the class under test. As mentioned in the threats to validity, we did a manual inspection of the results afterwards.

Next, we remove the class under test from the list of called classes and if the resulting list is empty the test is classified as a *unit test* otherwise it is a *component test*. To avoid misclassifying a unit test as a component test, we take two extra measures: (i) we remove super classes of the class under test because polymorphism is an accepted way to promote reuse between test classes. (ii) we remove the utility classes from the list of called classes. We identify the class as a utility class, if its name contains the word "Util". For example, consider the unit test case testCreateNumber in class NumberUtilsTest in project Lang (Bug ID 7b). The test case tests method createNumber(String) in class NumberUtils. The method createNumber(String), which turns the String into Number, first checks if the String is not empty or does not contain whitespace by calling isBlank(CharSequence) in utility class StringUtils. Without this extra step, the test would be misclassified as component test, since it calls class StringUtils besides the class under test NumberUtils.

Finally, we establish which kind of test exposes the fault. When *all* the failing tests are component tests, the defect is classified as exposed by component tests. When *all* the failing tests are unit tests, the defect is classified as exposed by unit tests. There are also a few cases where a defect is exposed by both a unit test and a component test. More precisely, for fault 15 in project Lang, one failing test is classified as a unit test while the other as a component test. Likewise, for fault 18 in project Chart, two failing tests are classified as unit tests while the other two as component tests. Since the fault in these two cases also involves component tests, which increase the search space, we classify the fault as exposed by component tests. This distinction resulted in a dataset totalling 73 faults exposed by unit tests and 273 faults exposed by component tests, however the distribution varies a lot over the projects. The Lang project has more unit tests, while the Closure project has mostly component tests. The particular details are presented in Table 5.1.

Further details about the algorithm are illustrated in Appendix A.

5.3.2 Evaluation Metrics

Fault localisation heuristics produce a ranked list of elements under test; in the ideal case the faulty unit appears on top of the list. Several ways to evaluate such rankings have been used in the past, including relative measures in relation to project size, such as the percentage of units that need or need not be inspected to pinpoint the fault [60]. However, absolute measures are currently deemed better for assessment purposes [60, 71]. The most commonly adopted metrics are *wasted effort*, *acc@n*, and *mean average precision* [8, 20,

60, 70]. Consequently, we will use these metrics for our evaluation.

To deal with faults spread over multiple locations, we evaluate from the perspective of a *best-case debugging* scenario as argued by Pearson et al. [85]. In such a scenario identifying one of the possible locations is good to understand and consequently repair the fault. Indeed, once the first faulty element is located it will help developers to find the remaining ones [8].

Mean Wasted Effort (MWE) — Smaller is better. The *mean wasted effort* is the mean of the *wasted effort* in all ranked lists. The *wasted effort* is an absolute measure which indicates the number of non-faulty methods to inspect in vain before reaching the first faulty method. It is computed as follows:

$$\text{wasted effort} = m + \frac{n}{2} \tag{5.1}$$

Where m is the number of non-faulty methods ranked strictly higher than the faulty method; and n is the number of non-faulty methods with equal rank in the ranked list to the faulty method. This deals with ties in the ranked list.

acc@n — Higher is better. This is the count of all the faults successfully localised in top-n positions in the ranked list. Inspired by B. Le et al., we also choose n ∈ {1, 3, 5}, thus effectively creating three variants of the *acc@n* namely *acc@1*, *acc@3*, and *acc@5* [20]. It is not uncommon for two methods in a ranked list sharing the same suspiciousness scores. Hence, while computing the *acc@n* precisely, we break the ties randomly.

Mean Average Precision (MAP) — Higher is better. The *mean average precision* has traditionally been used in information retrieval to evaluate the ranked lists and is also adopted for studying fault localisation. It takes all faulty elements into account and emphasises recall over precision. Thus, it is suitable in scenarios where developers search deep in the ranked list to find more relevant faulty elements [8]. The *mean average precision* is the mean of *average precision* in all ranked lists. The *average precision* in a single ranked list is calculated as following:

$$\text{average precision} = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{number\ of\ faulty\ methods} \tag{5.2}$$

Where: *i* indicates the position of a method in the ranked list; *M* is size of the ranked list (number of methods ranked); *pos*(*i*) is a boolean indicating whether or not the method at i^{th} position in the ranked list is faulty; *P*(*i*) is the precision at i^{th} position in the ranked list, computed as:

$$P(i) = \frac{\#\ faulty\ methods\ in\ top\ i}{i} \tag{5.3}$$

Our use of several metrics together evaluates fault localisation in several contexts. *Wasted effort* does not normalise the rank of faulty methods with respect to total number of methods in the program. Thus, it is inline with recommendations of Parnin and Orso in that for the fault localisation to be useful for developers the aim should be to improve absolute rank rather than percentage rank [71] . In their study, they found that developers switched to other means of debugging when they did not find faulty statements within the first few top positions in the ranked list. The same concerns are also addressed by *acc@n*. However, when developers want to search deep in the ranked list to find more relevant faulty methods, *mean average precision* is a more suitable metric [8]. Improving *mean average precision* may also imply the fault localisation to be useful for automated fault repair techniques [76]. These repair a fault by modifying potentially faulty program elements, starting from the top of ranked list, in a brute-force manner until a valid patch is generated.

5.3.3 Research Questions

The actual case study is driven by the following research questions.

RQ1. *Is the search space for component tests significantly larger than the one for unit tests?*

Motivation. Here we explore the underlying assumption of our evaluation: whether component tests indeed represent the challenging case for spectrum based fault localisation.

RQ2. *What is the best performing spectrum based fault localisation technique for the different projects?*

Motivation. This sets the baseline for the evaluation: identifying which of the sixteen techniques performs the best for the different projects in the dataset, irrespective of the composition of the test suite.

RQ3. *How well do spectrum based fault localisation techniques perform when the faults are exposed by unit tests?*

Motivation. For those faults exposed by unit tests (where we expect the search space to be rather small), we verify whether it is indeed true that these are the *easy* cases for fault localisation techniques.

RQ4. *How well do spectrum based fault localisation techniques perform when the faults are exposed by component tests?*

Motivation. Here we verify the performance for the *challenging* case: faults exposed by component tests. There we expect the search space to be large, i.e., all project

methods indirectly called by the test case.

RQ5. How long does it take to produce the ranked lists?

Motivation. This question addresses the inherent trade-off when choosing a spectrum based fault localisation: not only should they produce good rankings, but they should also do so within a reasonable time frame.

5.3.4 Evaluation Protocol

To evaluate the sixteen techniques under analysis, we first check out a faulty version for each project. Then, we run each spectrum based fault localisation for all triggering test classes, i.e. all test classes which trigger at least one of the project classes modified to fix the fault as recorded in the Defects4J dataset. As such, we obtain sixteen different ranked lists for each of the 346 faults in the dataset organised by project (see Table 5.1). We first evaluate the eight techniques within each spectrum analysis (Extended and Basic) before going into a more detailed analysis. We use five different metrics for this evaluation: Mean Average Precision (MAP), Mean Wasted Effort (MWE) and acc@1, acc@3, and acc@5.

Best Performing Fault Localisation Technique. We select the best performing technique from both spectrum analyses for each individual project, rather than aggregating the results over all the faults in the dataset. Selecting the best fault localisation technique on a project by project basis allows to observe how the techniques perform within one spectrum analysis and between the spectrum analyses. To achieve this, we first rank all sixteen techniques on each of the five evaluation metrics and then compute the mean of the ranks. Thus, the technique with the lowest mean rank performing best in all evaluation metrics for a given project is selected as the best performing one.

Tournament Ranking. Once the best fault localisation technique for a given project is known, we can compare the top techniques between the two spectrum analyses. Here we adopt the tournament ranking method with the five evaluation metrics [85]. That is, we set up a tournament structure for each evaluation metric for a given project, and award one point to the winner (either the technique from Extended or Basic spectrum analysis). The one with the highest *tournament score* performs better than its counterpart and is the overall best performing technique for a given project.

Significance Tests and Effect Size. We perform statistical tests of significance on the evaluation metrics *wasted effort* (*WE*) and *average precision* (*AP*). Since we have a matched pair design and we compare whether one variant is better than its counterpart, we choose the Wilcoxon signed rank test and run as paired one-tailed test. We favour the non-parametric Wilcoxon signed rank test over parametric t-test owing to small sample sizes and non-normal distribution of scores (and also their differences) for both *wasted effort* as

well as *average precision*. As per the common practice in software engineering research, we set the significance level α of 0.05 (there is 5% risk of concluding that the two distributions are different when in fact they are not). We measure the effect size with Cliff's delta [102] due to small sample sizes and non-normal distribution of scores (and also their differences) for both wasted effort as well as average precision.

However, for assessing the search space between unit tests and component tests where we do not have the matched pairs, we use the unpaired t-test and Cohen's d for tests of significance and effect size respectively.

Visualisation. All of these evaluations result in hundreds of results (5 projects $\times$ 2 spectrum analyses $\times$ 8 fault localisation techniques $\times$ 5 evaluation metrics $\times$ 2 comparisons). To explore all of these, we use kernel density plots as recommended by Kitchenham et al. [103]. These plots provide a detailed overview of the distributions, indicating the modality and densities. To compare the size of the search space for faults exposed by unit tests versus those exposed by component tests, we use violin plots [104].

Execution Time. While running the Basic spectrum analysis, the spectrum based fault localisation spends time to collect the method coverage during the execution of tests (tracing time) and to produce the ranked lists (ranking time). However, the Extended spectrum analysis induces an extra step: calculating the call pattern per method via the *closed itemset mining algorithm "CHARM"* [23]. This extra step (pattern mining) should be treated separately. Thus, while producing ranked lists we measure the tracing time and ranking time. When running Extended spectrum analysis, we also measure the pattern mining time. All the measurements are performed on a Mac machine (2.5 GHz Intel Core i7, 16 GB 1600 MHz DDR3) running MacOSX (10.11.6) using the bash internal variable $SECONDS (which indicates the number of seconds the script has been running) for a single run of the heuristic. As we run the tests sequentially, such time measurements are crude, thus should only be seen as an initial indicator for the relative time during the different steps.

Replication Package. The refinements we made to the Defects4J dataset and all the scripts as well as all the results of the evaluation are available at `https://github.com/glaghari/sbfl_unit_component_tests`.

5.4 RESULTS AND DISCUSSION

In this section, we address the four research questions introduced in Section 5.3.3.

RQ1. *Is the search space for component tests significantly larger than the one for unit tests?*

While separating failing tests into unit tests and component tests in Section 5.3.1, we make an assessment of the maximum size of search space. When the test case executes,

Table 5.3: Descriptive Statistics: Called Methods by Test Types

Test Type	Number of Tests	μ	σ
Unit tests	107	5.60	6.28
Component tests	656	1009.23	1192.53

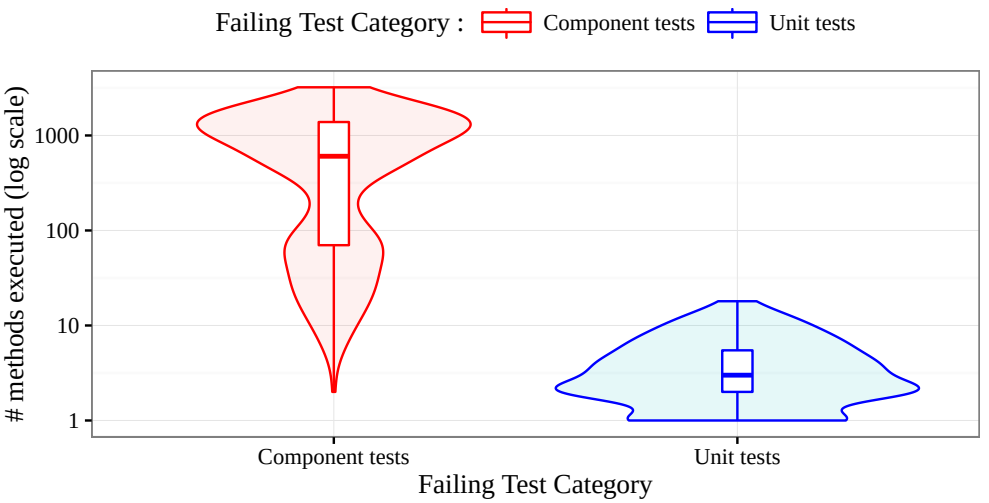


Figure 5.1: Assessment of the size of the search space for unit tests and component tests.

we count all project methods executed during the test case. Figure 5.1 shows the two distributions as violin plots. The Y-axis in the plot shows, on log scale, the total number of project methods executed in the failing test and the X-axis distinguishes between unit tests (blue) and component tests (red).

The red violin is higher and denser than its blue counterpart, indicating many more executed methods in component tests. As reported in Table 5.3, the mean (μ) as well as standard deviation (σ) of number of called project methods by component tests is considerably larger than the mean and standard deviation in category of unit tests. The t-test produces p-value 3.618647e-115 and Cohen's d 1.2, again confirming that the search space is significantly larger in failing tests categorised as component tests.

The size of the search space is significantly larger for component tests than for unit tests, confirming that the former is the challenging case for fault localisation techniques.

Table 5.4: Overall Scores of All Fault Localisation Techniques in Both Families for All Projects. The Values with Bold-face Indicate the Best Performing Technique.

Project	Family	Metric	Fault Localisation Technique							
			Barinel	D*	GP13	GP19	Op2	Ochiai	Tarantula	T*
Closure	E	acc@1	14	15	17	17	17	**16**	14	17
		acc@3	32	29	27	27	27	**31**	32	28
		acc@5	38	42	36	39	36	**40**	38	40
		MAP	0.1923160	0.1895809	0.1862639	0.1936689	0.1864209	**0.1924787**	0.1923025	0.1946540
		MWE	67.08	66.49	93.10	71.16	94.21	**66.80**	67.07	74.83
		Rank	4	2	5	3	5	**1**	4	2
	B	acc@1	5	5	**7**	6	7	5	5	6
		acc@3	12	11	**15**	13	15	11	12	14
		acc@5	16	16	**20**	19	20	16	16	20
		MAP	0.0850438	0.0889960	**0.1085065**	0.1019495	0.1084298	0.0874437	0.0850319	0.1034512
		MWE	246.72	225.09	**222.89**	293.14	222.89	225.95	246.52	221.78
		Rank	7	5	**1**	4	2	6	7	3
Math	E	acc@1	26	27	26	28	26	**29**	26	26
		acc@3	43	48	45	44	45	**47**	43	44
		acc@5	58	62	61	58	60	**63**	58	58
		MAP	0.3829402	0.4036051	0.3930052	0.3847035	0.3907057	**0.4054287**	0.3829402	0.3874786
		MWE	11.22	10.45	10.87	12.22	11.29	**10.58**	11.22	10.90
		Rank	7	2	3	6	4	**1**	7	5
	B	acc@1	22	**22**	21	21	21	21	22	21
		acc@3	43	**45**	43	44	43	44	43	44
		acc@5	49	**52**	51	50	51	51	49	50
		MAP	0.3293436	**0.3403478**	0.3298599	0.3246188	0.3295441	0.3301305	0.3293421	0.3290663
		MWE	18.42	**17.05**	17.49	22.21	17.53	17.29	18.42	17.50
		Rank	6	**1**	3	8	4	2	7	5
Lang	E	acc@1	32	37	37	**37**	37	37	32	37
		acc@3	46	50	51	**51**	51	49	46	51
		acc@5	50	51	53	**53**	53	52	50	53
		MAP	0.6524597	0.6855937	0.7061172	**0.7017472**	0.7075625	0.6832442	0.6524597	0.7005220
		MWE	3.60	2.92	2.83	**2.78**	3.72	2.93	3.60	2.82
		Rank	7	5	2	**1**	4	6	7	3
	B	acc@1	18	21	**21**	21	21	20	18	21
		acc@3	41	43	**43**	42	43	41	41	42
		acc@5	46	48	**49**	48	49	47	46	48
		MAP	0.4913956	0.5183098	**0.5238196**	0.5206148	0.5225065	0.5065799	0.4913956	0.5197306
		MWE	5.48	3.92	**3.78**	4.43	3.78	4.67	5.48	3.92
		Rank	5	3	**1**	3	2	4	5	3
Time	E	acc@1	5	9	7	8	7	**9**	5	8
		acc@3	7	10	9	9	9	**10**	7	10
		acc@5	11	14	13	14	13	**14**	11	14
		MAP	0.2629197	0.3367677	0.2901594	0.3063889	0.2901225	**0.3377602**	0.2629325	0.3100607
		MWE	39.12	31.62	34.54	31.35	42.00	**30.58**	39.12	31.08
		Rank	7	2	4	3	5	**1**	6	2
	B	acc@1	5	5	**5**	5	5	5	5	5
		acc@3	7	7	**8**	8	8	7	7	8
		acc@5	8	9	**9**	9	9	9	8	9
		MAP	0.2027214	0.2005875	**0.2201962**	0.2164297	0.2200860	0.1996957	0.2013044	0.2166605
		MWE	61.08	44.31	**39.38**	40.65	39.38	44.23	61.08	40.38
		Rank	5	5	**1**	4	2	5	6	3
Chart	E	acc@1	**8**	7	7	7	7	7	**8**	7
		acc@3	**13**	11	11	11	11	13	**13**	11
		acc@5	**16**	13	12	14	12	14	**16**	12
		MAP	**0.4190464**	0.4048134	0.3765929	0.3902784	0.3723341	0.4191233	**0.4190464**	0.3856748
		MWE	**11.48**	25.80	29.56	30.76	30.48	23.68	**11.48**	28.28
		Rank	**1**	3	5	4	6	2	**1**	4
	B	acc@1	10	10	9	9	9	10	**10**	9
		acc@3	15	12	11	11	11	13	**15**	11
		acc@5	16	14	13	13	13	15	**16**	12
		MAP	0.4985512	0.4740551	0.4298947	0.4278515	0.4298947	0.4831298	**0.4986104**	0.4243003
		MWE	27.16	37.04	39.84	40.52	39.84	35.08	**27.16**	39.24
		Rank	2	4	5	6	5	3	**1**	6

Table 5.5: Comparisons of the Two Families Showing the Top Ranked Techniques, the Tournament Scores, and the p-values Using All Faults. The Values with Bold-face Indicate One Variant is Significantly Better Than the Other.

Family	Project	Metric					Top Technique	T Score	p-values (δ)	
		acc@1	acc@3	acc@5	MAP	MWE			AP	WE
E	Closure	16	31	40	0.1924787	66.80	Ochiai_{E}	**P>R=5**	**5.66288e-08 (0.3)**	**1.413407e-12 (-0.3)**
	Math	29	47	63	0.4054287	10.58	Ochiai_{E}	**P>R=5**	**8.978418e-04** *(0.1)*	**3.671603e-06** (-0.2)
	Lang	37	51	53	0.7017472	2.78	GP19_{E}	**P>R=5**	**6.152051e-06** (0.3)	**5.063836e-04** (-0.2)
	Time	9	10	14	0.3377602	30.58	Ochiai_{E}	**P>R=5**	**1.033401e-02** (0.2)	**1.195428e-02** (-0.2)
	Chart	8	13	16	0.4190464	11.48	$\{\}_E$	P>R=1	—	—
	Overall	99	152	186	0.3714714	32.24	—	**P>R=5**	**1.413686e-12** (0.2)	**6.00831e-20** (-0.2)
B	Closure	7	15	20	0.1085065	222.89	GP13_{B}	R>P=0	—	—
	Math	22	45	52	0.3403478	17.05	D*_{B}	R>P=0	—	—
	Lang	21	43	49	0.5238196	3.78	GP13_{B}	R>P=0	—	—
	Time	5	8	9	0.2201962	39.38	GP13_{B}	R>P=0	—	—
	Chart	10	15	16	0.4986104	27.16	$\text{Tarantula}_{\text{B}}$	R>P=3	1.801039e-01 *(0)*	1.820786e-01 *(-0.1)*
	Overall	65	126	146	0.2861220	95.69	—	R>P=0	—	—

δ = Cliff's delta rounded to one decimal place indicating the effect size— **(medium)**, (small), and *(negligible)*.

$\{\}_E$ = $\text{Barinel}_{\text{E}}$, $\text{Tarantula}_{\text{E}}$

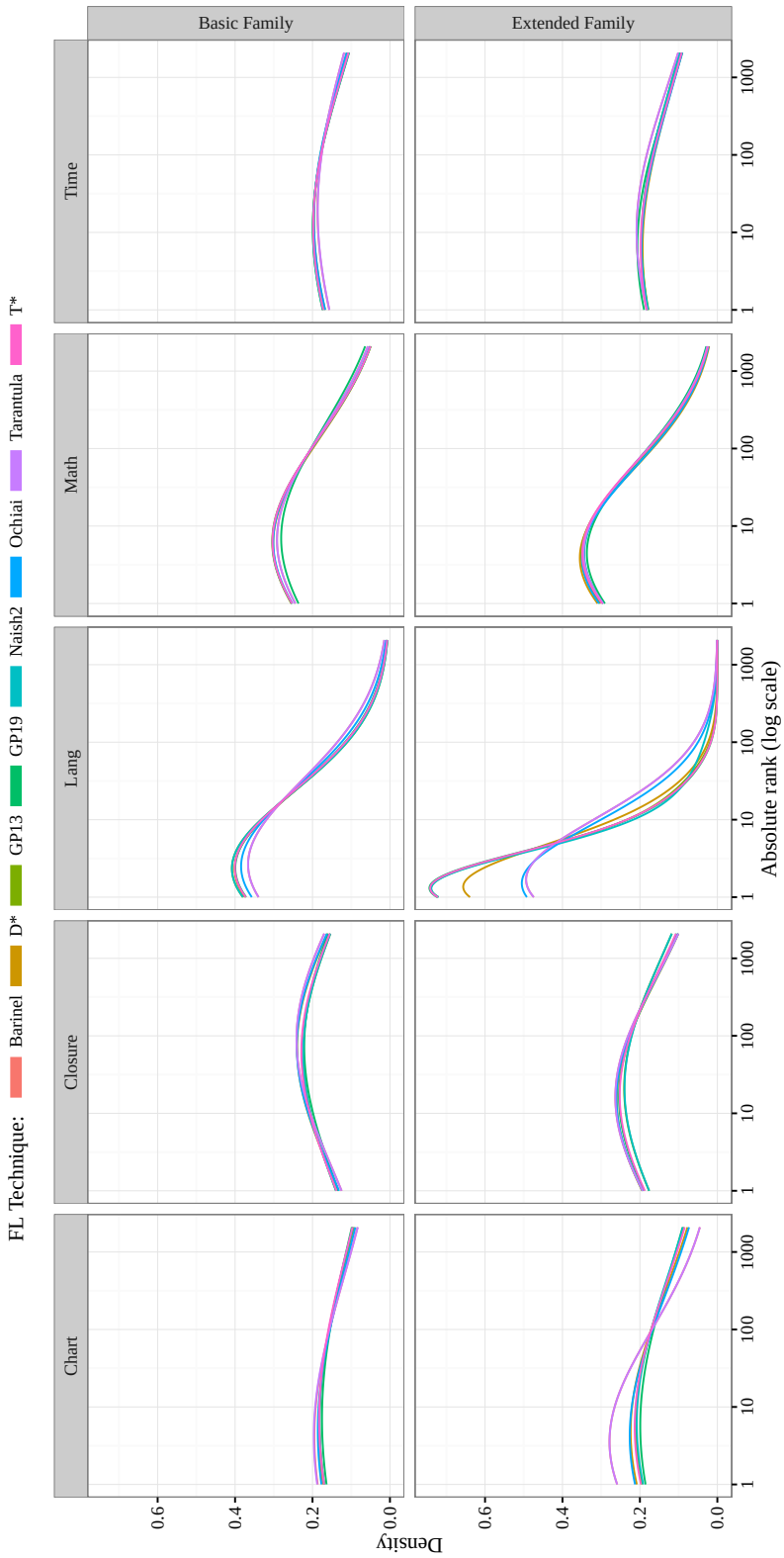


Figure 5.2: Distributions of absolute ranks of faulty methods for both spectrum analyses using all fault locators in a best-case debugging scenario for all fault categories.

RQ2. *What is the best performing spectrum based fault localisation technique for the different projects?*

As mentioned in the protocol, we rank the eight fault localisation techniques from both spectrum analyses for each project to determine the best performing fault localisation technique. We first do this for the complete test suite, ignoring the distinction between unit tests and component tests. Table 5.4 provides all the scores per project per spectrum analysis for the five evaluation metrics. With those we can determine the best performing technique: the rank is shown below the dashed line and the bold-faced column indicates the best performing one. Consider the Closure project as an example. For the Extended spectrum analysis, $\mathrm{Ochiai}_{\mathrm{E}}$ performs the best; yet $\mathrm{D^*}_{\mathrm{E}}$ is a close second since it has better scores for acc@5 and MWE. Similarly, for the Basic spectrum analysis the best performing technique is $\mathrm{GP13}_{\mathrm{B}}$; it has the best scores for all metrics. Furthermore, we notice in Table 5.5 that the $\mathrm{Ochiai}_{\mathrm{E}}$ and $\mathrm{GP13}_{\mathrm{B}}$ are dominant techniques in their respective spectrum analyses.

Furthermore, we also notice in Table 5.5 (row "Overall", column "acc@1") that with these top performing techniques, the techniques in the Extended spectrum analysis can successfully localise 99/346 ($\approx$ 29%) defects. While those in the Basic spectrum analysis can successfully localise 65/346 ($\approx$ 19%) defects.

Once the best performing technique is known for each spectrum analysis, we proceed by comparing the top techniques from both spectrum analyses against one another via tournament ranking and assess the statistical significance of the results. Table 5.5 provides the summary of comparisons for all projects. Consider again the Closure project, the tournament score for $\mathrm{Ochiai}_{\mathrm{E}}$ (E>B) is 5 meaning that it has better scores for all five evaluation metrics. In contrast, the tournament score for $\mathrm{GP13}_{\mathrm{B}}$ (B>E) is 0 meaning that none of five evaluation metrics scored better. We, therefore, can say that at least for the Closure project, $\mathrm{Ochiai}_{\mathrm{E}}$ is better than $\mathrm{GP13}_{\mathrm{B}}$. Moreover, the significance tests confirm that $\mathrm{Ochiai}_{\mathrm{E}}$ is also significantly better than $\mathrm{GP13}_{\mathrm{B}}$; the p-values shown in last columns of Table 5.5 are bold and the effect sizes (δ) are also medium.

The kernel density plots in Figure 5.2 help explore these comparisons in more detail. They show the distributions of absolute ranks of faulty method in the ranked lists for all techniques from both spectrum analyses. The absolute rank (X-axis) is the first position of any faulty method in the final ranked list. In the ideal ranking, the faulty method appears at the first position in the ranked list (rank 1). The Y-axis then shows the density of the corresponding ranks of faulty methods for all defects in a project. Higher densities for lower ranks are preferable for the rankings to be better; ideal is a density of 100% and peak for rank 1. Thus to interpret these plots, right-skewed curves are better than left-skewed ones.

For most projects, Figure 5.2 shows that the high densities and peaks for techniques from Extended spectrum analysis are more right-skewed than the corresponding ones from Basic spectrum analysis. However, the shape of the curves varies a lot over the projects, indicating that the choice of the best performing fault localisation technique is highly context dependent.

When running against the complete test suite, the best performing fault localisation techniques vary for the projects and within the spectrum analyses and so does their performance. Overall, the techniques in Extended spectrum analysis perform better than those from Basic spectrum analysis. The differences are significant yet the effect sizes are small.

RQ3. *How well do spectrum based fault localisation techniques perform when the faults are exposed by unit tests?*

Here, we evaluate the performance of fault localisation techniques on the faults exposed by unit tests, i.e. the easy cases. Due to space constraints we do not include all of the measurements used to determine the best performing fault localisation techniques (the details can be found in the replication package) and instead go immediately to the summary in Table 5.6. We point out, however, that here as well the best performing techniques varied over the projects and in some occasions even changed compared to the one used for all of the faults.

Given that the search space is smaller, the fault localisation techniques are expected to perform better and this is confirmed by the results. For both of the spectrum analyses, there are very high scores for mean average precision (a very strict measure and its scores are typically low [20]), acc@1, acc@3, and acc@5, and lower scores for mean wasted effort. Here, the fault localisation techniques from both spectrum analyses perform far better than previously (RQ2). Overall, the techniques from Extended spectrum analysis now successfully localise 45/73 ($\approx$ 62%) faults, with mean average precision 0.70 and mean wasted effort $\approx$ 4. Similarly, those in Basic spectrum analysis, now successfully localise 35/73 ($\approx$ 48%) faults, with mean average precision 0.64 and mean wasted effort $\approx$ 2.

Moreover, $\mathrm{GP13_E}$, $\mathrm{GP19_E}$, and $\mathrm{T^*_E}$ from Extended spectrum analysis significantly perform better than $\mathrm{GP19_B}$ for project Lang by successfully localising 30/40 faults. For the single defect in the Closure project, $\mathrm{GP19_E}$ successfully localises the fault, whereas none of the technique from Basic spectrum analysis do successfully localise the fault. In contrast, it is the opposite for the fault in the Time project. For the five faults in the Chart project, $\mathrm{GP19_E}$ only localises 1 fault. However, the four techniques from Basic spectrum analysis successfully localise 4 faults. Note that in the Closure, Time, and Chart projects

Table 5.6: Comparisons of the Two Families for Faults Revealed by Unit Tests.

Family	Project	Metric					Top Technique	T Score	p-values (δ)	
		acc@1	acc@3	acc@5	MAP	MWE			AP	WE
E	Closure	1	1	1	0.5021186	0.00	$GP19_E$	P>R=3	—	—
	Math	13	18	20	0.6018914	8.92	$Ochiai_E$	P>R=1	—	—
	Lang	30	37	38	0.8176481	2.23	$\{\}_{E1}$	**P>R=3**	**1.963428e-04** (0.3)	**2.099668e-03** (-0.2)
	Time	0	0	1	0.2500000	0.00	$\{\}_{E2}$	P>R=0	—	—
	Chart	1	2	4	0.3995238	5.60	$GP19_E$	P>R=0	—	—
	Overall	45	58	64	0.7000663	4.81	—	P>R=2	—	—
B	Closure	0	1	1	0.1709585	2.00	$\{\}_{B1}$	R>P=0	—	—
	Math	12	20	22	0.6192901	4.23	D^*_B	R>P=4	5.564735e-01 *(0)*	7.416792e-01 *(0)*
	Lang	18	35	38	0.6373810	1.52	$GP19_E$	R>P=0	—	—
	Time	1	1	1	1.0000000	0.00	$\{\}_{B2}$	R>P=4	—	—
	Chart	4	4	5	0.8500000	1.00	$\{\}_{B3}$	R>P=5	—	—
	Overall	35	61	67	0.6440786	2.44	—	R>P=3	9.953934e-01 (-0.2)	8.068516e-02 (0.2)

δ = Cliff's delta rounded to one decimal place indicating the effect size— **(medium)**, (small), and *(negligible)*.

$\{\}_{E1}$ = $GP13_E$, $GP19_E$, T^*_E.

$\{\}_{E2}$ = D^*_E, $Op2_E$, $GP13_E$, $Barinel_E$, $Ochiai_E$, $GP19_E$, $Tarantula_E$, T^*_E.

$\{\}_{B1}$ = D^*_B, $GP13_B$, $Barinel_B$, $Ochiai_B$, $Tarantula_B$, T^*_B.

$\{\}_{B2}$ = D^*_B, $Op2_B$, $GP13_B$, $Barinel_B$, $Ochiai_B$, $GP19_B$, $Tarantula_B$, T^*_B.

$\{\}_{B3}$ = D^*_B, $Op2_B$, $GP13_B$, $Ochiai_B$.

Table 5.7: Comparisons of the Two Families for Faults Revealed by Component Tests.

Family	Project	Metric					Top Technique	T Score	p-values (δ)	
		acc@1	acc@3	acc@5	MAP	MWE			AP	WE
E	Closure	15	30	39	0.1901151	67.31	Ochiai_{E}	**P>R=5**	**8.720041e-08 (0.3)**	**1.564843e-12 (-0.3)**
	Math	15	30	43	0.3411812	10.92	D*_{E}	**P>R=5**	**2.624127e-04** (0.3)	**3.925388e-08** (-0.3)
	Lang	7	14	15	0.4830555	4.05	$\{\}_{E1}$	**P>R=5**	**5.284793e-03 (0.4)**	**1.794968e-02** (-0.3)
	Time	9	10	13	0.3412707	31.72	Ochiai_{E}	**P>R=5**	**2.670496e-03** (0.3)	**1.036253e-02** (-0.3)
	Chart	7	11	13	0.4268924	12.50	$\{\}_{E2}$	P>R=4	5.411911e-01 *(0.1)*	6.219588e-02 *(0)*
	Overall	53	95	123	0.2853727	39.49	—	**P>R=5**	**4.168967e-12** (0.2)	**4.575609e-21** (-0.2)
B	Closure	7	14	19	0.1080297	224.58	GP13_{B}	R>P=0	—	—
	Math	10	25	30	0.2461595	21.38	D*_{B}	R>P=0	—	—
	Lang	3	8	11	0.2996969	8.25	Op2_{B}	R>P=0	—	—
	Time	4	7	8	0.1890040	40.96	GP13_{B}	R>P=0	—	—
	Chart	6	11	12	0.4204853	33.10	$\text{Tarantula}_{\text{B}}$	R>P=0	—	—
	Overall	30	65	80	0.1913367	120.58	—	R>P=0	—	—

δ = Cliff's delta rounded to one decimal place indicating the effect size— **(medium)**, (small), and *(negligible)*.

$\{\}_{E1}$ = GP13_{E}, GP19_{E} $\{\}_{E2}$ = $\text{Barinel}_{\text{E}}$, $\text{Tarantula}_{\text{E}}$

the samples are really small. Overall, the Basic spectrum analysis performs a little better than the Extended spectrum analysis, however the difference is insignificant with small effect size.

> *For both families, all techniques perform far better for faults exposed by unit tests only, thus confirming that such faults are the easy case for fault localisation.*

RQ4. *How well do spectrum based fault localisation techniques perform when the faults are exposed by component tests?*

Here as well, we omit the details for the selection of the best performing techniques and immediately move towards the summary in Table 5.7. For the faults exposed by component tests, the performance of the techniques from both spectrum analyses has decreased compared to the results in RQ2. The very same top performing techniques from Extended spectrum analysis, now can successfully localise 53/273 ($\approx$ 19%) faults. Also the mean average precision 0.29 is lower and the mean wasted effort $\approx$ 39 is higher. Likewise, for the Basic spectrum analysis, we can now only successfully localise 30/273 ($\approx$ 11%) faults. Here as well, the mean average precision 0.19 is lower and the mean wasted effort $\approx$ 121 is higher. Additionally, we observe that for project Lang $\mathrm{GP13_E}$ and $\mathrm{GP19_E}$ significantly perform better than $\mathrm{Op2_B}$ with effect size medium. Overall, the Extended spectrum analysis performs significantly better than the Basic spectrum analysis. Yet, the small effect size suggests that the practical differences between the two spectrum analyses are small.

Figure 5.3 juxtaposes the kernel density plots for unit tests and component tests which allows us to visually explore these differences in a bit more detail. Notably, the curves are more right-skewed with higher peaks for faults exposed by unit tests.

These observations indicate that when the techniques are evaluated on the whole dataset without distinction of the kind of tests involved, the better performance of fault localisation techniques largely follows from the easy-to-localise defects exposed by unit tests.

To asses the performance variation between the faults exposed by unit tests and component tests, we can compare the two proportions on acc@1 metric given by Extended spectrum analysis as it has significantly better results on faults exposed component tests. Overall, the Extended spectrum analysis successfully treats 45/73 ($\approx$ 62%) faults exposed by unit tests and 53/273 ($\approx$ 19%) faults exposed by component tests. At the 95% confidence level, the performance difference is 4% with confidence interval (30%–54%) and the 12% of margin of error.

Table 5.8: Summary of Analysis Times.

Family	Project	Time			
		Tracing	Sequence Generation	Ranking	Total
E	Closure	03:35:23	03:18:19	01:27:08	08:20:50
	Math	01:59:35	00:02:53	00:01:34	02:04:02
	Lang	00:20:54	00:00:51	00:00:38	00:22:23
	Time	00:12:05	00:00:57	00:00:44	00:13:46
	Chart	00:12:40	00:00:29	00:00:25	00:13:34
B	Closure	01:27:26	N/A	00:02:16	01:29:42
	Math	01:17:40	N/A	00:01:10	01:18:50
	Lang	00:15:13	N/A	00:00:38	00:15:51
	Time	00:09:37	N/A	00:00:18	00:09:55
	Chart	00:09:44	N/A	00:00:16	00:10:00

When evaluating against faults exposed by component tests, the performance of spectrum based fault localisation decreases from 30% to 54%. This confirms that component tests are the difficult case, and that the performance of a spectrum based fault localisation technique depends a lot on the presence of faults exposed by unit tests and component tests in the dataset.

RQ5. How long does it take to produce the ranked lists?

Table 5.8 provides an overview of the times for each of the phases in spectrum based fault localisation. We see that the execution times are mostly comparable although the Extended spectrum analysis is indeed slower. However, the Closure project is a notable outlier: $\approx 8\frac{1}{3}$ hours versus $1\frac{1}{2}$ hours. This can be explained by the observation that there are lots of tests in Closure project which call substantially more methods and thus take longer to analyse. However, as the defects in Closure are exposed by component tests (where the Extended spectrum analysis performs better) this long analysis time results in a better accuracy. Yet, the extra overhead for defects exposed by unit tests is not worthwhile.

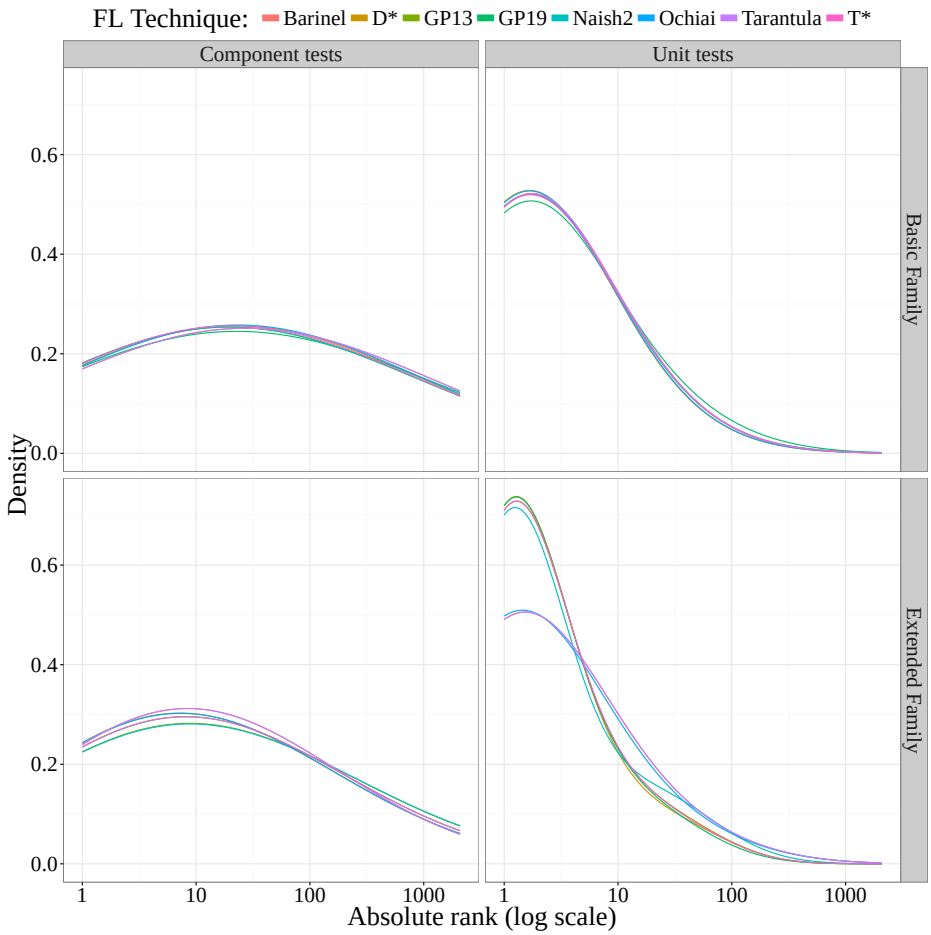


Figure 5.3: Distributions of absolute ranks of faulty methods for both spectrum analyses distinguishing between unit test and component test related faults.

The Extended spectrum analysis takes more time to produce the result, however the difference is negligible for smaller test suites. Contrary, the analysis time quickly grows for large tests and projects where Extended spectrum analysis comparatively takes more time than Basic implicating the risk of Extended spectrum analysis to become impractical.

5.5 THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent

with the guidelines for case studies research (see [80, 81]), we organise them into four categories.

Construct validity

Evaluation metrics. In this research, we evaluate sixteen different spectrum based fault localisation techniques. To reduce the risk on construct validity, we evaluated with five different metrics assessing different perspectives on what is deemed better. The use of an absolute metric (wasted effort) and also acc@n alleviates concerns on relative measures [71]. While the evaluation of fault localisation on mean average precision has implication for developers who search deep in the ranked list to find more relevant faulty methods and for automated fault repair techniques.

Separating tests into unit and component tests. Our heuristic to distinguishing between unit tests and component tests may misclassify certain tests compared to the intent of the original developers. We did a manual inspection of the results, hence the likelihood of misclassification is small. The results are available in the replication dataset and can be scrutinised by fellow researchers.

Internal validity

Method Level Granularity. We opted for method-level granularity for this comparison, mainly because there is evidence that this is the preferred by developers [87]. Nevertheless, the distinction between unit tests and component tests might not have such a big impact on coarse-grained granularity levels, in particular class level. Further analysis is needed to verify this.

Selection of Relevant Tests. In the case-study protocol, we run the spectrum based fault localisation for those test classes which trigger at least one of the source classes modified to fix the fault as recorded in the Defects4J dataset. However, in some cases the number of tests is relatively large. While we need to reduce test-suite size for increased efficiency (reduction in run-time) without compromising effectiveness (accuracy) of fault localisation, there is also an empirical evidence that the amount of tests used in the fault localisation has an effect (both positive and negative) on fault localisation effectiveness—there is a trade-off between test-suite reduction and fault localisation effectiveness [105]. How to strike the right-balance is an open issue in the spectrum based fault localisation community.

External validity

Several evaluation metrics together. We use several evaluation metrics together which implies a stringent comparison. Evaluation on a single metric alone may result in a different interpretations. A notable example in this chapter is comparison on project Chart (see Figure 5.2). If only evaluated on *Mean Wasted Effort*, the extended hit spectrum per-

forms better than the basic one. However, when comparing on several metrics together the result is different. This observation signals that the evaluation metric used to evaluate the fault localisation has an effect on its accuracy. Thus, it is unwise to generalise the findings, but instead one should value metric-specific insights.

Data-intensive and Mixed-language projects. In our study, we experimented with 346 real faults from Defects4J dataset; the most recent defect dataset currently available. Obviously, it remains to be seen whether similar results would hold for other defects in other systems. In particular, we don't know how well such fault localisation heuristics will work in data-intensive projects (where faults can sit in the data driving the application) or projects with complex technology stacks (e.g. mixing several programming languages and libraries).

Reliability and Verifiability

The external tool we rely upon is the open source library SPMF[1], for frequent itemset mining [106]. All of our scripts, results, etc. are available online in replication package (See Section 5.3.4) for reproducibility and verifiability (other researchers can confirm or refute the findings on other projects easily).

5.6 RELATED WORK

The Tarantula tool provided the foundation for research on spectrum based fault localisation [11]. Afterwards, several researchers made attempts to increase the effectiveness of spectrum based fault localisation, evaluating them differently including work on (a) finding the optimal *fault locators*, (b) changing the *hit-spectrum*, and (c) using machine learning to *learn to rank*. These state-of-the-art techniques spawned a new area of research called (d) *fault repair*, where the distinction between unit tests and component tests may also be relevant. Finally, the de facto standard dataset for this line of work is Defects4J, and we also discuss another extension thereof.

(a) Fault locators. Abreu et al. introduced Ochiai, used in the molecular biology domain, into spectrum based fault localisation and demonstrated better performance [67]. Steimann et al. defined and evaluated T* (which multiplies the suspiciousness given by Tarantula with confidence) and there as well demonstrated better performance [60]. Lucia et al. applied 20 well-known association measures from data mining on fault localisation and concluded that 10 out of 20 association measures were comparable to Tarantula and Ochiai [16]. Naish et al. proposed a couple of fault locators through a theoretical model and established that they performed better than existing ones [17]. Later studies confirmed that one (Op2) is among the best performing fault locators [20, 31, 85], which

[1] http://www.philippe-fournier-viger.com/spmf/

is corroborated in this study. Yoo evolved an entirely different set of fault locators (GP01–GP30) that performed better than existing ones [18]. Later, B. Le et al. found that GP13 and GP19 perform better [20].

(b) Hit-Spectra. Yilmaz et al. proposed time-spectrum as a spectrum based fault localisation variant exploiting a different hit spectrum [19]. Instead of coverage of methods, time-spectrum uses traces of method execution times collected from both passing and failing tests. The potential causes of faults are identified as deviations of failing tests from behaviour models created from time spectra collected in passing test runs. Laghari et al. proposed to extend the hit-spectrum, leveraging patterns of method calls by means of frequent itemset mining—and demonstrated the technique was more effective on Defects4J [31]. It is this latter variant that is used in this study.

(c) Learning to rank. Xuan and Monperrus proposed MULTRIC, a learning-based approach which combines multiple ranking metrics to learn and then rank [70]. They demonstrated on seeded faults that MULTRIC improved upon existing fault locators. Similarly, B. Le et al. proposed *Savant*—a learning to rank approach which exploited inferred likely method invariants mined from passing and failing test cases—and established that it was more effective on Defects4J [20]. Likewise, Sohn and Yoo have used code and change metrics (*age*, *churn*, *complexity*, ...) as features in learning to rank approach, establishing the technique is more effective on Defects4J [21].

(d)Test-suite reduction and diagnosability. Since the number of tests has an effect on the run-time of the spectrum based fault localisation, one of the improvements has been to reduce the test-suite size without sacrificing the fault localisation accuracy. Yu et al. performed the first of these investigations [105]. They found that the statement based reduction affected the fault localisation accuracy negatively, while the vector based reduction had negligible effects. Perez et al. propose a new metric, named DDU, with a goal to increase diagnosability of a test-suite in order to increase the effectiveness of spectrum based fault localisation [107]. In its preliminary evaluation, the optimisation of test-suites with DDU result in 34% gain in accuracy.

(e) Automatic Fault Repair is a recent research area aiming to automatically repair software faults [108]. Many of these techniques rely on fault localisation, in general, and spectrum based fault localisation, in particular, to identify the lines of code containing the fault and then applying a series of patches and running the test suite in the hope of obtaining a series of passing tests [109]. Initially these patches were generated in a brute-force manner, but recent approaches prune the amount of generated patches [110]. Our research complements this approach: rather than pruning the patch space our work suggests that pruning the search space focussing on faults exposed by unit tests may be a viable strategy. Note however that Martinez et al. demonstrated the critical role of the test suite, pointing out that many patches pass the test suite (thus result in a green verdict),

yet may still be incorrect [111].

> *Given that unit tests and component tests represent different strategies to pinpoint the location of a fault, one would expect that the research on fault localisation and fault repair also makes this distinction. However, even for the most recently proposed spectrum based fault localisation ([20, 21, 31, 98]) and fault repair techniques ([110, 111]) it is currently unknown how they deal with the more difficult faults exposed by component tests.*

Defects4J extensions. The main conclusion of our work is that researchers should distinguish between easy and difficult to locate faults when evaluating automatic fault localisation as well as fault repair techniques. To that extent, we extended Defects4J—the de facto standard dataset for this line of work with an extra qualitative property: an assessment of the type of test exposing the defect. Pertinent to mention here is also the recent work by Sobreira et al. which provides fine-grained details of the patches, especially the qualitative properties [112]. The availability of such fine-grained details about bug fixes within datasets opens opportunities to gain deeper and more actionable results.

5.7 CONCLUSION

In this chapter, we refined the Defects4J dataset distinguishing between unit tests and component tests. We established that the search space to locate the defect is rather small for defects exposed by unit tests (tests call fewer methods) thus represents the easy case, while the search space is significantly larger for defects exposed by component tests (tests call a lot of methods) thus represents the challenging case. Based on this distinction, we evaluated sixteen spectrum based fault localisation heuristics to see how they cope with defects exposed by unit tests or component tests. Not surprisingly, spectrum based fault localisation performs rather well for defects exposed by unit tests (acc@1 between 48% and 62%), however the performance decreases for defects exposed by component tests (acc@1 between 11% and 19%).

We have shown that the performance of spectrum based fault localisation heuristics depends a lot on the presence of faults exposed by unit tests or component tests in the dataset. This has an important consequence for future research in fault localisation. The evaluations with datasets containing lots of defects exposed by unit tests produce a biased view on the actionability of spectrum based fault localisation: they are very effective for easy to localise faults, but there the tool support hardly matters. Yet, for the harder to localise faults, the performance of spectrum based fault localisation still leaves a lot of

room for improvement. The good news is that the more recent approaches perform better on these more challenging cases, so there are viable avenues for future work.

Moreover, our work also has some implication with respect to automatic fault repair. It suggests that pruning the search space focussing on defects exposed by unit tests is a viable strategy. With this strategy, the fully automated fault repair techniques should focus on the low-hanging fruit and automatically repair the easy defects, this way freeing up resources and permitting humans to focus on the more difficult ones.

5.8 ACKNOWLEDGMENTS.

This work is sponsored by (a) the Higher Education Commission of Pakistan under a project titled “Strengthening of University of Sindh (Faculty Development Program)"; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry; (c) the Conseil Régional Hauts-De-France; Nord Pas de Calais — Picardie.

Chapter 6

Conclusions

Spectrum based fault localisation aids developers in locating the fault, once the tests detect the presence of those faults in the system. Since they only require the faulty program and the set of test cases that expose the fault, they are considered as lightweight. Spectrum based fault localisation techniques do not reason about the fault, rather pinpoint the fault by means of statistical analysis of the coverage information. While this simple process ideally suits to locate the faults when the tests fail, this is not always effective: many times the exact location of the faults is missed.

In this thesis, we attempted to increase the effectiveness of spectrum based fault localisation by exploring the use of *closed itemset mining* and *sequence mining*. By leveraging extra coverage information coupled with *closed itemset mining* and *sequence mining*, and analysing the patterns in the traces significantly improved the effectiveness of spectrum based fault localisation.

Additionally, we assessed the effectiveness of spectrum based fault localisation from a new perspective. We demonstrated that spectrum based fault localisation techniques were more effective on the faults exposed by unit tests, where the search space to locate the fault is sparse. Contrarily, the faults exposed by component tests represented the challenging cases for spectrum based fault localisation techniques manifested by their decreased effectiveness in pinpointing those faults. There the search space was comparatively vast, which implies that the techniques should be improved further to filter out the benign elements.

6.1 SUMMARY OF CONTRIBUTIONS

The main contributions of this thesis can be summarised as follows.

We proposed the use of *closed itemset mining* in spectrum based fault localisation. To that end, we first replicated a previous study, which used the sequence of method calls by sliding a window over the trace to locate the faulty classes [22]. We computed the sequence of method calls via *closed itemset mining* [23] and observed that *closed itemset mining* did boost the fault localisation effectiveness.

Next, we took the use of *closed itemset mining* from course-grained granularity *classes* to fine-grained granularity *methods*, to locate the faulty methods. Thus, we modified the hit-spectrum with the patterns of method calls via *closed itemset mining*. We demonstrated that the use of *closed itemset mining* increased the effectiveness of spectrum based fault localisation and remained more stable.

We also explored what is the effect of *sequence mining*, by replacing the *closed itemset mining* with *sequence mining* and observed that *sequence mining* also increased the effectiveness of spectrum based fault localisation. However, *sequence mining* became too expensive in terms of running time for large projects. Moreover, we analysed the performance of 47 fault locators and indicated the best performing ones on the Defects4J dataset.

Finally, we separated the faults in Defects4J into two categories; faults exposed by unit tests and faults exposed by component tests. We showed that, for the faults exposed by unit tests, the search space to locate the fault is smaller. Whereas, the search space was larger for faults exposed by component tests. We demonstrated that, spectrum based fault localisation techniques performed far better on faults exposed by unit tests than to faults exposed by component tests, which confirmed that the faults exposed by unit tests are comparatively easier to locate. On the other hand, the performance of spectrum based fault localisation techniques decreased when faced with the faults exposed by component tests, which confirmed that these are challenging cases and suggests that future fault localisation techniques should optimise for these difficult-to-locate faults.

6.2 SUMMARY OF RESEARCH QUESTIONS

This section briefly summarises the research questions explored in this thesis along with highlights of main findings.

RQ. Do the patterns of method calls extracted via closed itemset mining help boost the fault localisation accuracy in locating the faulty classes? In Chapter 2, we mainly explored this research question. We created a new fault localisation technique (SPEQTRA) to locate the faulty classes and replicated a previous study (AMPLE). We found that SPEQTRA performed significantly better than AMPLE owing to the use of itemset mining algorithm.

RQ3.1 Which ranking results in the lowest wasted effort: raw spectrum analysis or pat-

terned spectrum analysis? In Chapter 3, we modified the basic hit-spectrum (raw spectrum analysis) by incorporating the patterns of method calls obtained via itemset mining (patterned spectrum analysis) and compared the two experimentally. We found that for 68% cases, the patterned spectrum analysis ranked the faulty method higher than its counterpart raw spectrum analysis, implying a reduced wasted effort.

RQ3.2 How often do raw spectrum analysis and patterned spectrum analysis rankings result in a wasted effort $\leq$ 10? In Chapter 3, next in the comparison, we evaluated which of the two variants ranked the faulty method within a reasonable range of top locations (10). We found that patterned spectrum analysis ranked the faulty method in the top 10 for 62% of the cases against 48% for raw spectrum analysis.

RQ3.3 How does the number of triggered methods affect the wasted effort of raw spectrum analysis and patterned spectrum analysis? Finally in Chapter 3 we analysed the trend, the number of methods triggered and the position of faulty method in the ranked list, to measure the effect of the size of the ranked list over the rank of faulty methods. We found that patterned spectrum analysis was more stable while for raw spectrum analysis the faulty methods ranked lower (implying increased wasted effort) as the size of the ranked list increased.

RQ4.1 What is the baseline performance of raw spectrum analysis? In Chapter 4, we used 47 known fault locators for raw spectrum analysis to establish a baseline. We found that raw spectrum analysis placed the faulty method at top of the ranked list (acc@1) for 18% of the cases with the mean wasted effort 96.73.

RQ4.2 How much can sequenced spectrum analysis improve upon raw spectrum analysis? In Chapter 4, we modified the hit-spectrum via sequence mining algorithm where the patterns of method calls are both order-preserving and allow repetitive method calls, thus resulting into sequenced spectrum analysis. We found that compared to raw spectrum analysis, sequenced spectrum analysis gains 12% improvement for ranking the faulty method at top of the ranked list (acc@1) and reduces the average wasted effort from 96.73 to 25.88.

RQ4.3 Are there project specific differences between the rankings? In Chapter 4, we also compared the performance of the two variants on per project basis. We found that sequenced spectrum analysis performs better than raw spectrum analysis for four out of five projects. For the fifth project the results are practically indistinguishable.

RQ4.4 Is sequenced spectrum analysis efficient compared to raw spectrum analysis? Finally in Chapter 4 we gauged the timings to produce the ranked lists. We found that compared to raw spectrum analysis, sequenced spectrum analysis with an additional time overhead became very expensive for large project.

RQ5.1 Is the search space for component tests significantly larger than the one for unit tests? In Chapter 5, we added another perspective for assessment of spectrum based fault localisation, component test failures as challenging and unit test failures as easy cases for fault localisation. We confirmed this by establishing that the size of the search space to locate the fault is significantly larger for component tests than for unit tests.

RQ5.2 What is the best performing spectrum based fault localisation technique for the different projects? In Chapter 5, we constructed a suite of spectrum based fault localisation techniques to evaluate them for easy and challenging cases. We categorised the techniques into two families, Extended represented by patterned spectrum analysis and Basic represented by raw spectrum analysis. Here we established a baseline performance of the techniques on whole dataset irrespective of the distinction between easy and challenging cases. We also found that the performance of Extended spectrum analysis is better than those from Basic spectrum analysis significantly with small effect sizes.

RQ5.3 How well do spectrum based fault localisation techniques perform when the faults are exposed by unit tests? In Chapter 5, we confirmed that both families of spectrum based fault localisation perform far better for faults exposed by unit tests, thus such faults represent the easy cases.

RQ5.4 How well do spectrum based fault localisation techniques perform when the faults are exposed by component tests? Next, in Chapter 5, we confirmed that the performance of spectrum based fault localisation decreased a lot for faults exposed by component tests, thus such faults are indeed the difficult cases.

RQ5.5 How long does it take to produce the ranked lists? Finally, in Chapter 5, we also gauged the time taken by both families to output the ranked lists. We concluded that the time differences are negligible for smaller test suites, however, the Extended spectrum analysis comparatively takes more time than Basic for large tests and projects.

6.3 OUTLOOK

In the last decades, fault localisation has regained the attention of researchers: many new improved techniques have been proposed, user studies conducted showing their positive assistance in the debugging, etc. Yet, such techniques are still not widely adopted; the prevalent debugging method in developers is still the use of print statements [113]. One possible reason to account for why such automated fault localisation are seldom

used in practice can be that the prevalent evaluation strategy is quantitative in nature. The basis of the metrics used in such evaluations boils down to the number of faults in a dataset successfully treated by the techniques. Thus, the inherent premise of such evaluation strategy implies that there exists one-size-fits-all fault localisation technique, which we have empirically learned is not true.

Therefore, the fault localisation community should also focus in future fault localisation research to annotate the faults, such as we have annotated the faults as easy- or difficult-to-locate in Chapter 5. Similarly, the recent detailed fine-grained analysis of the patches of Defects4J by Sobreira et al. provides the opportunity to evaluate the fault localisation techniques to gain deeper and more actionable understanding of the techniques [112]. This should lead us to be able to link the effectiveness of the fault localisation with the properties of the tests, the coverage information, etc. so that we can choose a particular fault localisation technique for a certain scenario. This would not only aid practitioners in the debugging sessions, but also the fault repair researchers to improve the repair techniques, since automated fault repair depends on fault localisation.

Moreover, considering the study of Beller et al. [113], it appears that we are far from transferring automated fault localisation to industry. Thus, to learn the difficulties and the requirements of developers regarding fault localisation, we need more user studies—which up until now are very scarce. Therefore, the community should also consider user studies especially with the aim to learn how to transfer fault localisation research to industry.

Appendices

Appendix A

Defects4J Refinements

Here, first we elaborate the algorithm explained in Section 5.3.1 to classify the faults in Defects4J into either exposed by *unit tests* or *component tests*. Then, we illustrate the process with two representative examples; one example for each.

A.1 ALGORITHM TO CATEGORISE THE FAULTS

The algorithm explained in Section 5.3.1 is formally stated in Algorithm 1. In the input, T is the set of failing test cases for the faulty version of the project V_{fault}. For each failing test case t in the set T (Line 1 in Algorithm 1), the algorithm runs the test case t on the faulty version of the project V_{fault} to collect the classes C called during the execution of the test case (Line 3 in Algorithm 1). Along with the called class, the algorithm also collects the corresponding executed methods of the class.

Next, the algorithm iterates over the set of called classes C and determines the class under test c_t, which is the class with highest similarity score with the test case t (Lines 4–9 in Algorithm 1). Then, the class under test c_t, its super class, and *Utility* and *Mock* classes are removed from C (Lines 10–13 in Algorithm 1). Last, the test case t is classified as component test if C is *not* empty (Lines 14–15 in Algorithm 1), otherwise its marked as unit test (Lines 16–17 in Algorithm 1).

Finally, the algorithm categorises the faulty version of the project V_{fault} as related to Component tests if any of the failing test case t in T is classified as Component test (Lines 18–21 in Algorithm 1), otherwise categorises it as related to Unit tests (Lines 22–23 in Algorithm 1).

To determine the class under test c_t, the similarity score between the called class c and test case t is calculated in the procedure SimilarityScore(c, t). The score is determined by

Algorithm 1: Categorise Fault

```
   Input: V_fault, T
   Output: category
1  begin
2      for t ∈ T do
3          C ⟵ Coverage(t, V_fault)
4          c_t ⟵ C[0]
5          c_t.score ⟵ 0
6          for c ∈ C do
7              c.score ⟵ SimilarityScore(c, t)
8              if c.score > c_t.score then
9                  c_t ⟵ c
10         C.remove(c_t)
11         for c ∈ C do
12             if c.IsSuper(c_t or c.IsUtil() or c.IsMock()) then
13                 C.remove(c)
14         if |C| > 0 then
15             t.type ⟵ Component test
16         else
17             t.type ⟵ Unit test
18     for t ∈ T do
19         if t.type = Component test then
20             category ⟵ Component tests
21             return category
22     category ⟵ Unit tests
23     return category
```

Procedure SimilarityScore(c, t)

```
   Input: c, t
   Output: score
1  PackageScore ⟵ NameSimilarity(c.PackageName, t.PackageName)
2  ClassNameScore ⟵ NameSimilarity(c.Name, t.ClassName)
3  MethodNameScore ⟵ 0
4  for m ∈ c.Methods do
5      TmpMethodNameScore ⟵ NameSimilarity(m.Name, t)
6      if MethodNameScore ≤ TmpMethodNameScore then
7          MethodNameScore ⟵ TmpMethodNameScore
8  score ⟵ Sum(PackageNameScore, ClassNameScore, MethodNameScore)
9  return score
```

Table A.1: Called classes during the execution of failing test case in project Lang (Bug ID 6b)

Class Name	P	C	M	Sum
org.apache.commons.lang3.StringUtils	**1.00**	**1.00**	**0.00**	**2.00**
org.apache.commons.lang3.text.translate.UnicodeUnescaper	0.71	0.00	0.00	0.71
org.apache.commons.lang3.CsvEscaper	1.00	0.00	0.00	1.00
org.apache.commons.lang3.text.translate.OctalUnescaper	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.EntityArrays	0.71	0.00	0.20	0.91
org.apache.commons.lang3.CsvUnescaper	1.00	0.00	0.00	1.00
org.apache.commons.lang3.text.translate.LookupTranslator	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.UnicodeEscaper	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.CodePointTranslator	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.NumericEntityUnescaper	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.CharSequenceTranslator	0.71	0.00	0.00	0.71
org.apache.commons.lang3.text.translate.AggregateTranslator	0.71	0.00	0.00	0.71
~~org.apache.commons.lang3.StringEscapeUtils~~	1.00	0.67	0.25	1.92
~~org.apache.commons.lang3.ArrayUtils~~	1.00	0.33	0.00	1.33

the textual similarly between the package names, class names, and the test case name with method names of the class. The textual similarity is calculated with the metric *name similarity* [101]. The textual similarity between the package and class names of c and t is calculated in Lines 1 and 2 respectively. While the textual similarity between t and the highest matching method in c is calculated in Lines 4–7. The final similarity score between the called class c and test case t is the sum of matching scores of *package name*, *class name*, and *method name* (Line 8).

A.2 ILLUSTRATIVE EXAMPLES

Now, we can explain the process with a running example with the help of Algorithm 1. Consider the faulty version 6b of project Lang in Defects4J with one failing test case org.apache.commons.lang3.StringUtilsTest.testEscapeSurrogatePairs that exposes the fault. When running the test case, the collected called classes C are listed in Table A.1. The class org.apache.commons.lang3.StringUtils is determined as class under test, since it has highest matching score. Next, the algorithm removes the *Utility* classes (last two rows in Table A.1). Since there are still called classes remaining in the set C once the class under test and *Utility* classes are removed, the algorithm classifies the test case as component test. Finally, the algorithm categorises the fault 6b as exposed by component tests as there is only one failing test case, which is categorised as component test.

Similarly, consider the faulty version 7b of project Lang in Defects4J with one failing test case org.apache.commons.lang3.math.NumberUtilsTest.testCreateNumber that exposes the fault. The called classes C, collected during the execution of the test case, are listed in Table A.2. The class org.apache.commons.lang3.math.NumberUtils is determined as the

Table A.2: Called classes during the execution of failing test case in project Lang (Bug ID 7b)

Class Name	P	C	M	Sum
org.apache.commons.lang3.math.NumberUtils	**1.00**	**1.00**	**1.00**	**3.00**
~~org.apache.commons.lang3.SystemUtils~~	0.83	0.33	0.00	1.17
~~org.apache.commons.lang3.StringUtils~~	0.83	0.33	0.00	1.17

class under test , since it has the highest matching score. Next, the algorithm removes the remaining two *Utility* classes (last two rows in Table A.2). Since the set of called classes C is empty after the class under test and *Utility* classes are removed, the algorithm classifies the test case as a unit test Finally, the algorithm categorises the fault 7b as exposed by unit tests since there is only one failing test case, which is categorised as unit test.

Bibliography

[1] Gang Tan. A collection of well-known software failures. August 2016. URL `http://www.cse.psu.edu/~gxt29/bug/softwarebug.html`. (Cited on page 1).

[2] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology Report, May 2002. (Cited on page 1).

[3] Software fail watch: 5th edition. Tricentis Report, 2018. URL `https://www.tricentis.com/software-fail-watch/`. (Cited on page 1).

[4] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, September 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.314. URL `http://dx.doi.org/10.1109/MC.2005.314`. (Cited on page 1).

[5] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell – why researchers should care. In *Leaders of Tomorrow: Future of Software Engineering, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016. (Cited on pages 1 and 28).

[6] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. Automated debugging considered harmful considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 267–278, October 2016. doi: 10.1109/ICSME.2016.67. (Cited on page 1).

[7] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

URL http://dl.acm.org/citation.cfm?id=2337223.2337226. (Cited on pages 1 and 30).

[8] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, November 2013. ISBN 978-1-4799-0215-6. doi: 10.1109/ASE.2013.6693093. (Cited on pages 1, 30, 56, 62, 63, 79, 83, 84, and 85).

[9] Shivani Rao, Henry Medeiros, and Avinash Kak. Comparing incremental latent semantic analysis algorithms for efficient retrieval from software libraries for bug localization. *SIGSOFT Softw. Eng. Notes*, 40(1):1–8, February 2015. ISSN 0163-5948. doi: 10.1145/2693208.2693222. URL http://doi.acm.org/10.1145/2693208.2693222. (Cited on pages 1 and 30).

[10] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786880. URL http://doi.acm.org/10.1145/2786805.2786880. (Cited on pages 1, 30, 51, 52, and 79).

[11] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581397. URL http://doi.acm.org/10.1145/581339.581397. (Cited on pages 1, 30, 31, 32, 35, 56, 57, 66, 74, 79, 80, and 100).

[12] 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. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL http://doi.acm.org/10.1145/1101908.1101949. (Cited on pages 1, 22, 29, 30, 31, and 32).

[13] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82 (11):1780–1792, November 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.035. URL http://dx.doi.org/10.1016/j.jss.2009.06.035. (Cited on pages 1, 10, 12, 22, 29, 30, 31, 32, 39, and 56).

[14] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, August 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2521368. (Cited on pages 1 and 56).

[15] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2724-8. doi: 10.1109/PRDC.2006.18. URL `https://doi.org/10.1109/PRDC.2006.18`. (Cited on pages 2 and 74).

[16] Lucia, D. Lo, Lingxiao Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *2010 IEEE International Conference on Software Maintenance*, ICSM 2010, pages 1–10. IEEE, September 2010. doi: 10.1109/ICSM.2010.5609542. (Cited on pages 2, 31, 32, 57, 58, 74, 79, and 100).

[17] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL `http://doi.acm.org/10.1145/2000791.2000795`. (Cited on pages 2, 29, 56, 58, 66, 74, 80, and 100).

[18] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, pages 244–258, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33118-3. doi: 10.1007/978-3-642-33119-0_18. URL `http://dx.doi.org/10.1007/978-3-642-33119-0_18`. (Cited on pages 2, 58, 66, 74, 80, and 101).

[19] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell: Fault localization using time spectra. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 81–90, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368100. URL `http://doi.acm.org/10.1145/1368088.1368100`. (Cited on pages 2, 56, 58, 74, and 101).

[20] 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. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931049. URL `http://doi.acm.org/10.1145/2931037.2931049`. (Cited on pages 2, 56, 57, 58, 62, 63, 74, 78, 79, 80, 81, 83, 84, 93, 100, 101, and 102).

[21] 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. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092717. URL http://doi.acm.org/10.1145/3092703.3092717. (Cited on pages 2, 78, 79, 80, 101, and 102).

[22] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27992-X, 978-3-540-27992-1. doi: 10.1007/11531142_23. URL http://dx.doi.org/10.1007/11531142_23. (Cited on pages 2, 10, 14, 15, 16, 17, 22, 24, 25, 31, 57, 58, 60, 74, 79, and 106).

[23] Mohammed Javeed Zaki and Ching Jiu Hsiao. CHARM: an efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, pages 457–473, 2002. URL http://dx.doi.org/10.1137/1.9781611972726.27. (Cited on pages 3, 13, 17, 38, 87, and 106).

[24] Lionel Briand and Yvan Labiche. Empirical studies of software testing techniques: Challenges, practical strategies, and future research. *SIGSOFT Softw. Eng. Notes*, 29 (5):1–3, September 2004. ISSN 0163-5948. doi: 10.1145/1022494.1022541. URL http://doi.acm.org/10.1145/1022494.1022541. (Cited on page 4).

[25] Hwa-You Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442. IEEE Computer Society, 2008. ISBN 978-1-4244-2187-9. doi: 10.1109/ASE.2008.68. URL http://dx.doi.org/10.1109/ASE.2008.68. (Cited on page 5).

[26] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 141–152. ACM, 2009. ISBN 978-1-60558-338-9. doi: 10.1145/1572272.1572290. URL http://doi.acm.org/10.1145/1572272.1572290. (Cited on page 5).

[27] D. Lo, H. Cheng, and X. Wang. Bug signature minimization and fusion. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, pages 340–347, Nov 2011. doi: 10.1109/HASE.2011.36. (Cited on page 5).

[28] Chengnian Sun and Siau-Cheng Khoo. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 576–586. ACM, 2013. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491449. URL http://doi.acm.org/10.1145/2491411.2491449. (Cited on page 5).

[29] Zhiqiang Zuo, Siau-Cheng Khoo, and Chengnian Sun. Efficient predicated bug signature mining via hierarchical instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 215–224. ACM, 2014. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610400. URL http://doi.acm.org/10.1145/2610384.2610400. (Cited on page 5).

[30] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Localising faults in test execution traces. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, IWPSE 2015, pages 1–8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3816-5. doi: 10.1145/2804360.2804361. URL http://doi.acm.org/10.1145/2804360.2804361. (Cited on pages 7, 31, 56, 57, 58, 74, and 79).

[31] 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. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970308. URL http://doi.acm.org/10.1145/2970276.2970308. (Cited on pages 7, 56, 57, 58, 78, 79, 80, 81, 100, 101, and 102).

[32] Gulsher Laghari and Serge Demeyer. On the use of sequence mining within spectrum based fault localisation. In *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*, SAC 2018, 2018. ISBN 978-1-4503-5191-1/18/04. doi: 10.1145/3167132.3167337. URL https://doi.org/10.1145/3167132.3167337. (Cited on page 7).

[33] Gulsher Laghari and Serge Demeyer. Poster: Unit tests and component tests do make a difference on fault localisation effectiveness. In *Proceedings of the 40th International Conference on Software Engineering Companion*, ICSE-C '18, 2018. ISBN 978-1-4503-5663-3/18/05. doi: 10.1145/3183440.3194970. URL https://doi.org/10.1145/3183440.3194970. (Cited on page 7).

[34] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. (Cited on pages 9 and 28).

[35] A. Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE '08. Conference*, pages 289–293, Aug 2008. doi: 10.1109/Agile.2008.8. (Cited on pages 10 and 28).

[36] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87(0):48 — 59, 2014. (Cited on pages 10 and 28).

[37] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4), 2006. (Cited on page 10).

[38] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the Int'l Conference on Automated Software Engineering (ASE)*, pages 433–444. IEEE CS, 2009. (Cited on pages 10 and 28).

[39] Andy Zaidman, Bart Van Rompaey, van Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011. (Cited on pages 10 and 28).

[40] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006. (Cited on pages 10 and 28).

[41] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.60. URL `http://dx.doi.org/10.1109/ASE.2009.60`. (Cited on pages 10, 23, 24, and 75).

[42] Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, March 2013. ISSN 1049-331X. doi: 10.1145/2430536.2430541. URL `http://doi.acm.org/10.1145/2430536.2430541`. (Cited on page 10).

[43] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005. (Cited on pages 10 and 55).

[44] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252 – 2268, 2008. doi: http://dx.doi.org/10.1016/j.jss.2008.02.068. (Cited on page 13).

[45] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1597-5. URL `http://dl.acm.org/citation.cfm?id=647883.738238`. (Cited on pages 14, 22, and 66).

[46] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005. ISSN 1382-3256. doi: 10.1007/s10664-005-3861-2. URL http://dx.doi.org/10.1007/s10664-005-3861-2. (Cited on pages 15, 25, 31, and 78).

[47] Jingxuan Tu, Lin Chen, Yuming Zhou, Jianjun Zhao, and Baowen Xu. Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 1–8, Aug 2012. doi: 10.1109/QSIC.2012.30. (Cited on pages 22 and 24).

[48] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006. (Cited on page 23).

[49] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806806. URL http://doi.acm.org/10.1145/1806799.1806806. (Cited on page 23).

[50] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2013. (Cited on page 24).

[51] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635910. URL http://doi.acm.org/10.1145/2635868.2635910. (Cited on page 28).

[52] Martin Fowler and Matthew Foemmel. Continuous integration (original version). http://http://www.martinfowler.com/, September 2010. Accessed: April, 1st 2016. (Cited on page 28).

[53] N. Tillmann and W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.117. (Cited on page 28).

[54] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint*

Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 179–190. ACM, 2015. (Cited on page 28).

[55] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 471–482, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818813. (Cited on pages 28 and 32).

[56] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287632. URL http://doi.acm.org/10.1145/1287624.1287632. (Cited on pages 28 and 58).

[57] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818815. (Cited on page 28).

[58] Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. Understanding the impact of rapid releases on software quality: The case of firefox. *Empirical Software Engineering*, 20(2):336–373, 2015. (Cited on page 28).

[59] René Just, Darioush Jalali, and Michael 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, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2628055. URL http://doi.acm.org/10.1145/2610384.2628055. (Cited on pages 28, 32, 41, 56, 61, 78, and 82).

[60] 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, pages 314–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483767. URL http://doi.acm.org/10.1145/2483760.2483767. (Cited on pages 29, 30, 31, 39, 43, 56, 57, 62, 66, 74, 79, 80, 81, 83, 84, and 100).

[61] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130, Nov 2012. doi: 10.1109/ISSRE.2012.28. (Cited on pages 30, 31, 39, 43, 51, and 56).

[62] Nicholas DiGiuseppe and James A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 210–220, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001446. URL http://doi.acm.org/10.1145/2001420.2001446. (Cited on pages 30 and 52).

[63] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 1105–1112, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: 10.1145/1143844.1143983. URL http://doi.acm.org/10.1145/1143844.1143983. (Cited on page 30).

[64] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351752. URL http://doi.acm.org/10.1145/2351676.2351752. (Cited on pages 31, 32, 57, 79, and 80).

[65] Jingxuan Tu, Lin Chen, Yuming Zhou, Jianjun Zhao, and Baowen Xu. Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs. In *Proceedings of the 2012 12th International Conference on Quality Software*, QSIC '12, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4833-3. doi: 10.1109/QSIC.2012.30. URL http://dx.doi.org/10.1109/QSIC.2012.30. (Cited on pages 31, 57, and 79).

[66] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *J. Syst. Softw.*, 89:51–62, March 2014. ISSN 0164-1212. doi: 10.1016/j.jss.2013.08.031. URL http://dx.doi.org/10.1016/j.jss.2013.08.031. (Cited on pages 31 and 32).

[67] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2984-4. URL http://dl.acm.org/citation.cfm?id=1308173.1308264. (Cited on pages 31, 32, 56, 57, 58, 66, 74, 79, 80, and 100).

[68] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642983. URL http://doi.acm.org/10.1145/2642937.2642983. (Cited on pages 31, 32, 35, 57, and 79).

[69] Tien-Duy B. Le, David Lo, and Ferdian Thung. Should i follow this fault localization tool's output? *Empirical Softw. Engg.*, 20(5):1237–1274, October 2015. ISSN 1382-3256. doi: 10.1007/s10664-014-9349-1. URL http://dx.doi.org/10.1007/s10664-014-9349-1. (Cited on pages 31, 32, 57, and 79).

[70] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME 2014, pages 191–200. IEEE, September 2014. doi: 10.1109/ICSME.2014.41. (Cited on pages 31, 43, 57, 62, 74, 79, 84, and 101).

[71] 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. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001445. URL http://doi.acm.org/10.1145/2001420.2001445. (Cited on pages 31, 51, 56, 62, 63, 75, 83, 85, and 99).

[72] Pragya Agarwal and Arun Prakash Agrawal. Fault-localization techniques for software systems: A literature review. *SIGSOFT Softw. Eng. Notes*, 39(5):1–8, September 2014. ISSN 0163-5948. doi: 10.1145/2659118.2659125. URL http://doi.acm.org/10.1145/2659118.2659125. (Cited on pages 31 and 78).

[73] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X. URL http://dl.acm.org/citation.cfm?id=257734.257766. (Cited on page 31).

[74] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321702. URL http://doi.acm.org/10.1145/1321631.1321702. (Cited on pages 31 and 78).

[75] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337225. (Cited on page 32).

[76] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, January 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2560811. (Cited on pages 32 and 85).

[77] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483785. URL http://doi.acm.org/10.1145/2483760.2483785. (Cited on page 32).

[78] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9. (Cited on pages 35, 57, 77, and 79).

[79] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 460–469, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-609-7. doi: 10.1145/1281192.1281243. URL http://doi.acm.org/10.1145/1281192.1281243. (Cited on page 51).

[80] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering*, 14(2):131–164, 2009. (Cited on pages 51, 75, and 99).

[81] Robert K. Yin. *Case Study Research: Design and Methods, 3 edition*. Sage Publications, 2002. (Cited on pages 51, 75, and 99).

[82] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013. doi: 10.1109/ASE.2013.6693093. (Cited on pages 51 and 52).

[83] Xiaozhen Xue and Akbar Siami Namin. How significant is the effect of fault interactions on coverage-based fault localizations? In *2013 ACM / IEEE International*

Symposium on Empirical Software Engineering and Measurement, pages 113–122, October 2013. (Cited on page 52).

[84] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 141–150, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6146-7. doi: 10.1109/ICSME.2014.36. URL http://dx.doi.org/10.1109/ICSME.2014.36. (Cited on page 52).

[85] 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. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.62. URL https://doi.org/10.1109/ICSE.2017.62. (Cited on pages 57, 62, 74, 78, 79, 80, 81, 84, 86, and 100).

[86] Jonathan Aldrich Joshua Sushine, James D. Herbsleb. Searching the state space: A qualitative study of api protocol usability. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2015. (Cited on page 58).

[87] 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. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931051. URL http://doi.acm.org/10.1145/2931037.2931051. (Cited on pages 58, 79, and 99).

[88] Boris Cule, Nikolaj Tatti, and Bart Goethals. Marbles: Mining association rules buried in long event sequences. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(2):93–110, 2014. (Cited on pages 60, 73, and 76).

[89] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Trans. Softw. Eng.*, 39(5):613–637, May 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.63. URL http://dx.doi.org/10.1109/TSE.2012.63. (Cited on page 60).

[90] Andreas Zeller, Thomas Zimmermann, and Christian Bird. Failure is a four-letter word: A parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 5:1–5:7, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0709-3. doi: 10.1145/2020390.2020395. URL http://doi.acm.org/10.1145/2020390.2020395. (Cited on pages 64 and 69).

[91] Alberto González Sánchez. Automatic error detection techniques based on dynamic invariants. Master's thesis, Delft University of Technology, the Netherlands, 2007. URL http://swerl.tudelft.nl/twiki/pub/Main/AlbertoGonzalezSanchez/thesis_gonzalez.pdf. (Cited on page 66).

[92] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.25. URL http://dx.doi.org/10.1109/ASE.2009.25. (Cited on pages 66 and 80).

[93] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 449–456, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. doi: 10.1109/COMPSAC.2007.109. URL http://dx.doi.org/10.1109/COMPSAC.2007.109. (Cited on page 66).

[94] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806806. URL http://doi.acm.org/10.1145/1806799.1806806. (Cited on page 74).

[95] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 591–600, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985874. URL http://doi.acm.org/10.1145/1985793.1985874. (Cited on page 75).

[96] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2009. (Cited on pages 77 and 82).

[97] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, August 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2521368. (Cited on page 78).

[98] 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. ISBN 978-1-4503-5076-1. doi:

10.1145/3092703.3092731. URL http://doi.acm.org/10.1145/3092703.3092731. (Cited on pages 79, 80, and 102).

[99] 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. ISSN 0018-9529. doi: 10.1109/TR.2013.2285319. (Cited on page 80).

[100] Joep Weijers. Extending project lombok to improve junit tests. Master's thesis, Delft University of Technology, the Netherlands, 2012. URL http://repository.tudelft.nl/islandora/object/uuid:1736d513-e69f-4101-8995-4597c2a4df50/datastream/OBJ/download. (Cited on page 82).

[101] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *J. Syst. Softw.*, 88:147–168, February 2014. ISSN 0164-1212. doi: 10.1016/j.jss.2013.10.019. URL http://dx.doi.org/10.1016/j.jss.2013.10.019. (Cited on pages 82 and 115).

[102] Norman Cliff. Answering ordinal questions with ordinal data using ordinal statistics. *Multivariate Behavioral Research*, 31(3):331–350, 1996. doi: 10.1207/s15327906mbr3103_4. PMID: 26741071. (Cited on page 87).

[103] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630, 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9437-5. URL http://dx.doi.org/10.1007/s10664-016-9437-5. (Cited on page 87).

[104] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998. ISSN 00031305. URL http://www.jstor.org/stable/2685478. (Cited on page 87).

[105] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368116. URL http://doi.acm.org/10.1145/1368088.1368116. (Cited on pages 99 and 101).

[106] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. Spmf: A java open-source pattern mining library. *J. Mach. Learn. Res.*, 15(1):3389–3393, January 2014. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2627435.2750353. (Cited on page 100).

[107] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *ICSE 2017, Proceedings*

of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, May 2017. (Cited on page 101).

[108] Martin Monperrus. Automatic software repair: A bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, January 2018. ISSN 0360-0300. doi: 10.1145/3105906. URL http://doi.acm.org/10.1145/3105906. (Cited on page 101).

[109] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3990-4. doi: 10.1109/ICST.2010.66. URL http://dx.doi.org/10.1109/ICST.2010.66. (Cited on page 101).

[110] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486893. (Cited on pages 101 and 102).

[111] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, August 2017. ISSN 1382-3256. doi: 10.1007/s10664-016-9470-4. URL https://doi.org/10.1007/s10664-016-9470-4. (Cited on pages 101 and 102).

[112] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Monperrus Martin, and Marcelo Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, SANER 2018, page to appear, March 2018. URL http://saner.unimol.it/accepted. (Cited on pages 102 and 109).

[113] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *International Conference on Software Engineering*, ICSE 2018, May-June 2018. doi: 10.1109/ICSME.2016.67. (Cited on pages 108 and 109).