Methods and Measures for Statistical Fault Localisation

David Landsberg

Linacre College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2016

This thesis is dedicated to
my parents, Anthony and Kathryn Landsberg,
for all their support

Acknowledgements

I would like to thank my supervisors Daniel Kroening (Oxford University) and Hana Chockler (King's College London) for their guidance, and the Oxford Formal Verification group for many helpful discussions. In particular, I would like to thank my office mate Matt Lewis for his help in many areas. I would like to thank Friedrich Steimann (University of Hagen), Marcus Rui Abreu (University of Porto), Lucia Lucia (University of Luxembourg), and Alex Groce (Oregon State) for making their benchmarks available to me for the purposes of my experiments – it is only as a function of their repositories that this thesis was able to provide results for what is (to my knowledge) one of the largest scale experiments performed in software fault localisation to date. Finally, I would like to thank Ganesh Narayan, Alison Prata, and Gemma Prata for many helpful comments on this thesis.

As Steiman et al. remark, experimentation in this field is "an arduous undertaking" [213], and the main body of this thesis does not acknowledge the large amount of time spent preparing and re-performing experiments of our scale. Echoing Steimann et al., it does not "count the many repeats of experiments that became necessary because we picked the wrong performance indicators, did the wrong aggregation, or because we had discovered bugs in our programs" [213]. Finally, this thesis does not acknowledge the large amount of time spent developing new fault localisation methods which which did not, in the end, make the final cut.

Publications and Submissions

Some chapters of this thesis describes and develops work from the following three papers which were completed during the course of this thesis. The first has been published, the second has been accepted for publication and is being presented at the forthcoming IBM conference in Haifa, and the third is under review.

- David Landsberg et al.: *Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme*. Foundations of Automated Software Engineering (FASE) 2015: 115-129 (published).
- David Landsberg et al.: *Probabilistic Fault Localisation*. Haifa Verification Conference (HVC) 2016 (accepted for publication).
- David Landsberg et al.: *Spectrum Based Fault Localisation of Single and Multiple Faults: New Measures and a Multiple Fault Localisation Optimiser*. Journal of Automated Software Engineering (ASE) (submitted).

Abstract

Fault localisation is the process of finding the causes of a given error, and is one of the most costly elements of software development. One of the most efficient approaches to fault localisation appeals to statistical methods. These methods are characterised by their ability to estimate how faulty a program artefact is as a function of statistical information about a given program and test suite. However, the major problem facing statistical approaches is their effectiveness – particularly with respect to finding single (or multiple) faults in large programs typical to the real world.

A solution to this problem hinges on discovering new formal properties of faulty programs and developing scalable statistical techniques which exploit them. In this thesis I address this by identifying new properties of faulty programs, developing the formal frameworks and methods which are formally proven to exploit them, and demonstrating that many of our new techniques substantially and statistically significantly outperform competing algorithms at given fault localisation tasks (using $p = 0.01$) on what (to our knowledge) is one of the largest scale set of experiments in fault localisation to date.

This research is thus designed to corroborate the following *thesis statement*: That the new algorithms presented in this thesis are effective and efficient at software fault localisation and outperform state of the art statistical techniques at a range of fault localisation tasks. In more detail, the major *thesis contributions* are as follows:

1. We perform a thorough investigation into the existing framework of (SBFL), which currently stands at the cutting edge of statistical fault localisation. To improve on the effectiveness of SBFL, our first contribution is to introduce and motivate many new statistical measures which can be used within this framework. First, we show that many

are well motivated to the task of SBFL. Second, we formally prove equivalence properties of large classes of measures. Third, we show that many of the measures perform competitively with the existing measures in experimentation – in particular our new measure $m9185$ outperforms all existing measures on average in terms of effectiveness, and along with Kulkzynski2, is in a class of measures which statistically significantly outperforms all other measures at finding a single fault in a program ($p = 0.01$).

2. Having investigated SBFL, our second contribution is to motivate, introduce, and formally develop a new formal framework which we call probabilistic fault localisation (PFL). PFL is similar to SBFL insofar as it can leverage any suspiciousness measure, and is designed to directly estimate the probability that a given program artefact is faulty. First, we formally prove that PFL is theoretically superior to SBFL insofar as it satisfies and exploits a number of desirable formal properties which SBFL does not. Second, we experimentally show that PFL methods (namely, our measure PFL-PPV) substantially and statistically significantly outperforms the best performing SBFL measures at finding a fault in large multiple fault programs ($p = 0.01$). Furthermore, we show that for many of our benchmarks it is theoretically impossible to design strictly rational SBFL measures which outperform given PFL techniques.

3. Having addressed the problem of localising a *single* fault in a program, we address the problem of localising *multiple* faults. Accordingly, our third major contribution is the introduction and motivation of a new algorithm $M_{Opt(g)}$ which optimises any ranking-based method g (such as PFL/SBFL/Barinel) to the task of multiple fault localisation. First we prove that $M_{Opt(g)}$ formally satisfies and exploits a newly identified formal property of multiple fault optimality. Secondly, we experimentally show that there are values for g such that $M_{Opt(g)}$ substantially and statistically significantly outperforms given ranking-based fault localisation methods at the task of finding multiple faults ($p = 0.01$).

4. Having developed methods for localising faults as a function of a given test suite, we finally address the problem of optimising test

suites for the purposes of fault localisation. Accordingly, we first present an algorithm which leverages model checkers to improve a given test suite by making it satisfy a property of single bug optimality. Second, we experimentally show that on small benchmarks single bug optimal test suites can be generated (from scratch) efficiently when the algorithm is used in conjunction with the CBMC model checker, and that the test suite generated can be used effectively for fault localisation.

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Basic Concepts . . . 3
1.3 Problem and Thesis Statement . . . 5
1.4 Thesis Contributions . . . 6
1.5 Thesis Outline . . . 8

2 Literature Review **9**

2.1 Early Approaches . . . 9
2.1.1 Manual approaches . . . 9
2.1.2 Slicing approaches . . . 10
2.2 Reasoning-based Approaches . . . 11
2.2.1 Program state-based approaches . . . 12
2.2.2 Model-based approaches . . . 12
2.3 Spectrum-based Approaches . . . 14
2.3.1 Developing measures . . . 16
2.3.2 Defining spectra . . . 20
2.3.3 Improving test suites . . . 21
2.4 Multiple-Technique Combination . . . 24
2.5 Multiple-Fault Localisation Approaches . . . 24
2.6 Research Directions . . . 27

3 Preliminaries **29**

3.1 Probands . . . 29
3.2 Proband Models . . . 31
3.3 Program Spectra . . . 35
3.4 Probability Spaces . . . 36
3.5 Suspiciousness Measures . . . 39

3.6 Properties . . . 39
3.7 SBFL method . . . 41

4 Experimental setup **43**
4.1 Benchmarks . . . 43
4.1.1 Small programs . . . 44
4.1.2 Large programs . . . 47
4.2 Evaluation Methods . . . 51
4.2.1 Wasted effort scores . . . 51
4.2.2 Absolute ranking scores . . . 54
4.2.3 Upper and lower bound scores . . . 55
4.2.4 Higher level scores . . . 56
4.2.5 Significance tests . . . 57
4.2.6 Evaluation . . . 57
4.3 Threats to Validity . . . 58

5 Spectrum-based Fault Localisation **60**
5.1 New Measures . . . 61
5.1.1 Similarity measures . . . 62
5.1.2 Prediction measures . . . 66
5.1.3 Causal measures . . . 68
5.1.4 Confirmation measures . . . 71
5.1.5 Automatically generated measures . . . 74
5.2 Equivalence Proofs . . . 76
5.3 Empirical Evaluation . . . 81
5.3.1 Scores by average . . . 81
5.3.2 Scores by number of faults . . . 86
5.3.3 Discussion . . . 86
5.4 Summary . . . 96

6 Probabilistic Fault Localisation **97**
6.1 Motivating Example . . . 98
6.2 Assumptions . . . 99
6.3 PFL-Equations . . . 103
6.4 Measure of Causal Propensity . . . 105
6.5 Properties . . . 106
6.6 PFL-Algorithm . . . 108

6.7 Empirical Evaluation . . . 110
6.7.1 Scores by average . . . 111
6.7.2 Scores by number of faults . . . 112
6.7.3 Discussion . . . 112
6.8 Summary . . . 117

7 Multiple-fault Localisation **118**
7.1 Motivating Example . . . 119
7.2 When to stop searching for faults . . . 121
7.3 MFL Algorithm . . . 123
7.4 Properties . . . 127
7.5 Empirical Evaluation 1: Optimised SBFL and PFL . . . 129
7.5.1 Setup . . . 129
7.5.2 Scores by average . . . 131
7.5.3 Scores by number of faults . . . 132
7.5.4 Discussion . . . 132
7.6 Empirical Evaluation 2: Optimised Barinel . . . 138
7.6.1 Setup . . . 139
7.6.2 Results . . . 140
7.7 Summary . . . 142

8 Test-suite Optimisation **144**
8.1 Properties . . . 145
8.2 Algorithm . . . 147
8.3 Example . . . 150
8.4 Experimentation . . . 153
8.4.1 Setup . . . 153
8.4.2 Results . . . 154
8.5 Summary . . . 156

9 Conclusions **157**
9.1 Thesis Summary . . . 157
9.2 Future Work . . . 160
9.2.1 Tool development and usability studies . . . 160
9.2.2 Other applications . . . 160

Bibliography **162**

A Appendix **187**
A.1 Established SBFL Measures 187
A.2 SBFL and PFL W-scores 193
A.3 MFL W-scores 196
A.4 PFL Case Study 199

List of Figures

3.1 `minmax.c` 30
3.2 Basic probabilistic expressions and their measures 38

5.1 Performance of some prominent measures 85
5.2 W-scores for selected measures on SIR Benchmarks 87
5.3 W-scores for selected measures on Steimann Benchmarks 87
5.4 A-scores for selected measures on SIR Benchmarks 88
5.5 A-scores for selected measures on Steimann Benchmarks 88

6.1 `minmax.c` 99
6.2 coverage matrix 99
6.3 Performance of prominent PFL techniques 111
6.4 Performance of PFL-PPV on Steimann benchmarks 112
6.5 W-scores for selected techniques on SIR benchmarks 113
6.6 W-scores for selected techniques on Steimann benchmarks 113
6.7 A-scores for selected techniques on SIR benchmarks 114
6.8 A-scores for selected techniques on Steimann benchmarks 114

7.1 `minmax2.c` 120
7.2 W-scores for selected techniques on SIR Benchmarks 133
7.3 W-scores for selected techniques on Steimann Benchmarks 133
7.4 Number of faults found on SIR Benchmarks 134
7.5 Number of faults found on Steimann Benchmarks 134
7.6 Performance of SBFL techniques optimised and unoptimised 135
7.7 W-scores for selected techniques on SIR benchmarks 141
7.8 W-scores for selected techniques on Steimann benchmarks 141

8.1 `minmax.c` preprocessed 151

A.1 Fault probabilities for eventbus-1357981291647 201

List of Symbols

$\boldsymbol{PM}$	program model, an ordered set of components
$\mathbf{T}$	set of test cases
$\mathbf{P}$	set of passing test cases
$\mathbf{F}$	set of failing test cases, where $\mathbf{T} = \mathbf{P} \cup \mathbf{F}$
C_i	component, where $C_i \in \boldsymbol{PM}$ and $0 < i \leq \lvert\boldsymbol{PM}\rvert$
E	error component, where $E = C_{\lvert\boldsymbol{PM}\rvert}$
t_k	test case, where $t_k \in \mathbf{T}$
c_i^k	Boolean which is true just in case C_i is covered by test case t_k, where $c_i^k \in \mathbb{B}$
a_{ef}^i	number of failing traces covering C_i
a_{nf}^i	number of failing traces that don't cover C_i
a_{ep}^i	number of passing traces covering C_i
a_{np}^i	number of passing traces that don't cover C_i
w	suspiciousness measure, where $w : \boldsymbol{PM} \to \mathbb{R}$
$\mathbf{H}$	set of fault hypotheses
h_i	the hypothesis that C_i is faulty, where $h_i \in \mathbf{H}$
$\mathbf{C}$	set of causal hypotheses
h_i^k	the hypothesis that C_i was the cause of E in t_k, where $h_i^k \in \mathbf{C}$

Chapter 1

Introduction

1.1 Motivation

A 2013 study at the University of Cambridge's Judge Business School reports that the process of *debugging* (defined as the process of finding and fixing a fault) costs the global economy up to US$312 billion dollars a year as a function of wasted programmer time [223]. The costs to the U.S. economy alone make up a sizeable proportion of the economic problem – according to a 2002 study by the Department of Commerce's National Institute of Standards and Technology (NIST), "Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated US$59.5 billion annually, or about 0.6 percent of the gross national product" [221, 265]. Many researchers add that this "cost has undoubtedly grown since then" due to the growing ubiquity and complexity of software [235]. Thus, the existence of faults in software has created a major economic problem.

In addition to the general costs to the world economy, failure to successfully debug software has been single-handedly responsible for major catastrophes and fatalities. Limiting ourselves to three prominent examples, the following cases are now well established in software verification lore [13, 133]:

1. In 1996 the control software of Ariane 5 malfunctioned, causing the explosion of the spaceship 37 seconds after launch – destroying the rocket and payload valued at US$500 million [65, 159].

2. In 1998 the Mars Climate Orbiter built by Nasa's Jet Propulsion laboratory (JPL) approached Mars at the wrong angle due to different units of measurement being used in the codebase – ruining a US$327.6 million project in minutes [12].

3. During the period of 1985–7, the Therac-25 radiation therapy machine administered massive drug overdoses which was caused by a software upgrade which in turn allowed the operator to type faster when entering data. However, the parts of the machine that were not upgraded could not cope with the speed of forthcoming data entries and caused the machine to administer the wrong doses, resulting in the deaths of patients [186].

In summary, buggy software has a dramatic effect on the global economy, has caused major newsworthy catastrophes, and has compromised human safety [1]. For these reasons the project of improving the debugging process, so that software contains fewer faults upon delivery, is imperative – promising to reduce costs, avert disaster, and save lives.

How, then, can we improve the debugging process? A first question to ask is why software is released with bugs in the first place. As no programmer is perfect, industrial programs will often have bugs introduced into them at the development stage, where the debugging process suffers from two known pitfalls:

1. The debugging process is often a time consuming and laborious process – estimated to consume 50–60% of the time a programmer spends in the maintenance and development process [51].

2. The debugging process often fails; in the sense that it often fails to remove faults – with software commonly being released even though they are known to contain them [2].

These problems are no doubt explained by the fact that debugging is notoriously difficult. In fact, some have suggested it is *too* difficult for a human; Brian Kernighan (co-inventor of the C programming language) writes: "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever

[1] see [186] for discussion of many of these sorts of disasters) Reports of software failures are now commonplace in the news due to the general public nuisance they cause. For example, see http://www.computerworlduk.com/galleries/infrastructure/top-10-software-failures-of-2014-3599618/

[2] https://www.theguardian.com/technology/2006/may/25/insideit.guardianweeklytechnologysection

as you can be when you write it, how will you ever debug it?" [138]. To overcome the difficulties associated with human performed debugging, it is then proposed that we automate parts of the debugging process in order to make debugging quicker and more successful. This would mean engineers would be able to find and repair more bugs in faulty programs in improved time scales – helping to minimise the problems associated with faulty software overall.

How then can we automate parts of the debugging process in order to achieve this? In more detail, the phrase "to debug" is defined by the IEEE standard as "to detect, locate, and correct faults in a computer program" [11]. We may elaborate on each these three stages as follows:

1. In the first *bug detection* (or *testing*) phase, the program is tested as to whether it satisfies a given specification or not. If the code is shown to violate the specification (i.e. shown to work incorrectly) then there must be a bug in the code.

2. In the second *fault-localisation* phase, the bug that was confirmed to exist in the testing phase is located.

3. In the third *correction* (or *repair*) stage, the successfully located bug is corrected/repaired.

To improve debugging, we might improve methods involved at any of its three stages. Of the three stages, the *fault localisation* stage has been consistently reported as the most time consuming, costly, and difficult for the user [109, 225]. Thus, any methods which can improve the fault localisation stage has the potential to alleviate the problems associated with debugging overall. The project of this thesis is to provide such a solution to this problem.

The rest of this chapter is organised as follows. We introduce the basic concepts underlying software fault localisation in 1.2, describe the software fault localisation problem and our thesis statement in 1.3, and summarise our contributions in 1.4. Finally, an outline of the structure of this thesis is presented in 1.5.

1.2 Basic Concepts

In this section we informally define concepts which are fundamental to the software fault-localisation problem. The Institute of Electrical and Electronics Engineers

(IEEE) provide a standard glossary of software engineering terminology, which includes many definitions relevant to software fault localisation [11]. We shall use their terminology throughout the thesis, which includes the following definitions:

- *Mistake.* A human action that produces an incorrect result.
- *Fault.* A fault can either be
  1. A defect in a hardware device or component; for example, a short circuit or broken wire.
  2. An incorrect step, process, or data definition in a computer program.
- *Error.* The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.
- *Failure.* The inability of a system or component to perform its required functions within specified performance requirements.

We use the term "fault" and "bug" synonymously. Following the current literature in software fault localisation [235], the type of bugs focused on are assumed to have a more fine grained definition than those provided for by the IEEE. Firstly, all the faults considered limited to software faults by definition. Secondly, the faults considered are best described as "Bohrbugs" as opposed to "Mandelbugs" [235]. The technical distinction is as follows [107]:

- *Bohrbug.* A fault that is easily isolated and that manifests consistently under a well-defined set of conditions. (Here, easily isolated means easy to verify as a bug, as opposed to easy to find).
- *Mandelbug.* A bug whose underlying causes are so complex and obscure as to make its behaviour appear chaotic and even non-deterministic.

To illustrate the difference: A handful of lines of code with incorrect insertions or deletions (aka omissions) is considered a Bohrbug. A key feature of Bohrbugs is that they are easily identifiable as bugs – indeed this is a necessary condition for making fault-localisation experiments on benchmarks possible. In contrast, we would consider a massive-scale convolution of ill-understood pointer networks which cause

an error as constituting a Mandelbug - here the causes and nature of the bug are potentially so obscure as to defy a clear understanding or rectification.

Two desirable properties of any fault localisation technique are as follows (here informally described).

- *Effectiveness.* The degree of effectiveness of a fault-localisation method can be measured in a variety of ways, and is often determined as a function of how much non-faulty code the user has to inspect before the user finds a bug using the method (measures of wasted programmer effort [13]), or alternatively of whether the method correctly identifies a fault within the first given number of guesses [188].

- *Efficiency.* The degree of efficiency of a fault-localisation method is measured in terms of the expected time/memory resources it will consume in order to present fault localisation information to the user.

An ideal fault localisation method is expected to have high effectiveness and efficiency.

1.3 Problem and Thesis Statement

We use the definitions of the previous section to informally describe the software fault localisation problem. A programmer makes a *mistake* when he/she writes an incorrect line of code in software, where this line (or lines) constitutes the *fault* (*Bohrbug*). These faults cause *failure/s* in the system, where the difference between the failed output and the correct output is the event of the *error*. With this understanding, the problem of software fault localisation can be stated as follows:

- *Problem Statement.* The problem of software fault localisation is the problem of finding effective and efficient algorithms for localising faults in software.

Accordingly, this thesis is designed to advance solutions to the software fault localisation problem by corroborating the following thesis statement:

- *Thesis Statement.* The new algorithms presented in this thesis are effective and efficient at software fault localisation and experimentally outperform state of the art statistical techniques at a range of fault localisation tasks.

Here, our new algorithms include new spectrum based fault localisation (SBFL) measures, a new framework of probabilistic fault localisation (PFL), and a new multiple fault localisation optimiser to be used with SBFL and PFL techniques. The state of the art statistical techniques compared against include all known SBFL techniques and BARINEL (discussed in Chapter 2). The range of fault localisation tasks include single and multiple fault localisation, and out-performance includes statistically significantly better performance ($p = 0.01$) using given effectiveness measures.

1.4 Thesis Contributions

In this section we discuss our major contributions in further detail. Our major *thesis contributions* follow a general formula of firstly identifying hitherto undiscovered formal properties of faulty programs, and then developing lightweight statistics based methods which satisfy and exploit them at the given fault localisation task.

1. In Chapter 5, we perform a thorough investigation into the existing framework of (SBFL), which currently stands at the cutting edge of statistical fault localisation. To improve on the effectiveness of SBFL, our first contribution is to introduce and motivate many new statistical measures which can be used within this framework. First, we show that many are well motivated to the task of SBFL. Second, we formally prove equivalence properties of large classes of measures. Thirdly, we show that many of the measures perform competitively with the existing measures in experimentation – in particular our new measure $m9185$ outperforms all existing measures on average in terms of effectiveness, and along with Kulkzynski2, is in a class of measures which statistically significantly outperforms all other measures at finding a single fault in a program ($p = 0.01$).

2. Having performed a thorough investigation into SBFL, the major contribution of Chapter 6 is to introduce, motivate, and formally develop a new formal framework which we call probabilistic fault localisation (PFL). PFL is similar to SBFL insofar as any given suspiciousness measure can be leveraged, and is designed to directly estimate the probability that a given program artefact is faulty. First, we formally prove that PFL is theoretically superior to SBFL insofar as it satisfies and exploits a number of desirable formal properties which SBFL does not. Second, we experimentally show that PFL methods (namely, our measure PFL-PPV)

substantially and statistically significantly ($p = 0.01$) outperforms the best performing SBFL measures at finding a fault in our large multiple fault programs. Furthermore, we show that for many of our benchmarks it is theoretically impossible to design strictly rational SBFL measures which outperform given PFL techniques.

3. Having addressed the problem of localising a *single* fault in a program, in Chapter 7 we address the problem of finding *multiple* faults. Accordingly, our third major contribution is the introduction and motivation of a new algorithm $M_{Opt(g)}$ which optimises any ranking based (such as PFL/SBFL/BARINEL) to multiple fault localisation. First we prove that $M_{Opt(g)}$ formally satisfies and exploits a newly identified formal property of *multiple fault optimality*. Second, we experimentally show that the algorithm substantially and statistically significantly ($p = 0.01$) improves given ranking-based fault localisation methods at our multiple fault localisation tasks.

4. Having developed methods for localising faults as a function of a given test suite, we finally address the problem of generating test suites themselves. Accordingly, we first present an algorithm which leverages model checkers to generate small test suites which satisfy a property of single bug optimality. Second, we experimentally show that on small benchmarks single bug optimal test suites can be generated efficiently when the algorithm is used in conjunction with the CBMC model checker.

Each contribution is designed to follow up on the corresponding research directions presented in the literature review in Section 2.6. Along with the above contributions, we emphasise that our results are supported by, what is to our knowledge, the largest scale experimentation in fault localisation to date. This is the largest in several dimensions. Firstly, in the number of program versions (50k+) experimented upon. Secondly, in the number of techniques compared (almost 200 manually defined SBFL measures, and thousands of automatically generated measures). Thirdly, in the range of bugs (1 to 32). This larger scale comparison allows us to make more robust conclusions about fault localisation techniques compared.

1.5 Thesis Outline

The rest of this thesis is organised as follows. In Chapter 3 we present the formal preliminaries underlying the approaches considered in this thesis. In Chapter 4.2 we present our experimental set-up common to many of the experiments performed in this thesis. In Chapter 5 we present our work in SBFL. In Chapter 6 we present our new framework of PFL. In Chapter 7 we present our new multiple fault localisation algorithm. In Chapter 8 we present a method for optimising test suites by appeal to model checkers. Finally in Chapter 9 we conclude by summarising our contributions and give directions for future work.

Many chapters of this thesis describe and develop work published or submitted to conferences and journals, as follows. Chapter 5 describes and develops work presented in Landsberg et al. [149]. Major contributions not included in that paper which are included in the aforementioned chapter include a substantial increasing of the number of suspiciousness measures compared, and a substantial expansion into the size and scale of the experimentation, along with identification of new best performing measures as determined. Chapter 6 describes and develops work presented in Landsberg et al. [148]. Major contributions not included in that paper which are included in the aforementioned chapter include formal presentation of the PFL algorithm (informally described in the paper), experimental results which improve on the average runtime of PFL-PPV from an average of 6 seconds to 1.80 thanks to an improved implementation, and detailed experimental results from additional datasets. Chapter 7 describes and develops work presented in a third (submitted) paper. Major contributions not included in that paper which are included in the aforementioned chapter include using the PFL techniques described in Chapter 6 in conjunction with the proposed multiple fault localisation technique.

Chapter 2

Literature Review

In this chapter we review the current state of the art in software fault localisation. The purpose of the review is to identify and motivate the research problems that this thesis addresses. As our aim is to find a solution to the problem of software fault localisation in general (i.e. it can be used, in principle, for a wide range of programs and languages et cetera), we shall focus on general methods, and therefore avoid specific applications. We have focused on approaches which have traction in the literature and have been reported widely elsewhere [235, 237]. For a recent review of software fault localisation methods the reader is referred to the survey of Wong et al. [235].

2.1 Early Approaches

In this section we briefly review manual debugging techniques, along with the first steps to introducing some automation to the process.

2.1.1 Manual approaches

The following methods are amongst the earliest approaches to fault localisation, and may be described as "manual" insofar as they require the user to be constantly involved at each step of the debugging process. This includes the following methods:

1. *Retracing*, in which the user retraces the steps of a failing trace in order to evaluate which part of the program caused the error [48].

2. *Logging*, in which the user inserts print statements across the code. Executions of the program will then print-out program-state information which can be

stored in logs and inspected to give the user more understanding about how the error was caused [68].

3. *Assertions*, in which the user inserts assertion statements across the code. Assertions are constraints that the developer can use to specify what the system is designed to do, and will raise an exception if violated in an execution. This exception can be used to detect incorrect program behaviour and localise the fault [201, 202].

4. *Breakpoints*. A breakpoint transfers control of an execution to the user when a specified point in the program is reached. At these breakpoints the user can inspect and manipulate a given program state, or perform a step-by-step investigation of the execution. Debugging tools which incorporate this feature include GNU-GDB [4] and the Microsoft Visual Studio Debugger [6].

5. *Profiling*. Profiling gathers profiles of a given execution, such as memory usage and the frequency and duration of function calls, to be used for analysis. Tools that use profiling for debugging include GNU's GPROF [5] and Eclipse's TPTP [9].

Traditional "manual" methods are generally accepted to be time consuming and laborious to perform [225]. Consequently, researchers have looked to either semi-automated or fully automated methods to improve the process in terms of efficiency and effectiveness. The remainder of this literature review focuses on such methods.

2.1.2 Slicing approaches

Program slicing is a widely published and surveyed area [36, 222, 248], and provides one of the oldest fault-localisation methods dating back to Weiser's 1979 article [232]. When applied to software fault localisation, the aim of a slicing approach is to categorise parts of a given program or execution deemed causally irrelevant to a given error. What remains is called the program/execution slice, and aids in fault localisation as the engineer's search space is potentially reduced. Slicing methods can be broadly categorised into the following classes: static slices, dynamic slices, execution slices, and dices:

1. *Static slices*. A static slice for a given variable-statement pair contains all the executable statements that could possibly affect the value of the variable at the given program statement. Static slices are found by computing consecutive sets of indirectly relevant statements using data and control flow dependencies [187, 231] or information flow relations [82].

2. *Dynamic slices.* A dynamic slice for a variable-statement-value triple contains all the executable statements that could possibly affect that particular value of this variable at the statement. They are thus potentially smaller than static slices because fewer parts of a program will satisfy the more restrictive conditions [2, 25, 26, 28, 61]. [1]

3. *Execution slices.* An execution slice can be understood as a dynamic/static slice where the domain of the program has been restricted to a particular set of inputs (and therefore a single execution). They are thus a lot less expensive to compute than the above types of slices, as the slice is only computed for a given set of inputs, and not the whole program [23]. Execution slice tools include χ-Suds at Telcordia [22, 81] and eXVantage [157].

4. *Dices.* A program dice is the difference between two sets of slices of failing/passing traces [167]. Slices and dices can be combined to improve fault localisation potential.

Outstanding problems with slicing techniques include their efficiency and effectiveness at fault-localisation. With respect to effectiveness, slices can be "lengthy and hard to understand" [235]. In addition, the fault localisation information can be too coarse grained insofar as it only presents two sets of information (those in/out of the slice). With respect to efficiency, many dynamic slicing methods take up excessive time and file space [235], with past attempts trying to limit these problems [35, 260].

At the state of the art, it is common to see slicers integrated into more advanced fault localisation techniques, such as spectrum based techniques (see [131,236]), model based approaches (see [103]), and reasoning approaches (see [243]). They can consequently be understood as providing a good additional stage in many different software fault localisation algorithms, as opposed to being an advanced stand-alone fault-localisation method in itself. As such, slicing methods are still much researched, with 20% of the software fault localisation literature estimated to be on slicing topics [235].

2.2 Reasoning-based Approaches

Reasoning-based approaches localise faults by using automated reasoning and modelling methods to check that desired properties (specifications) are satisfied by the program under test. They constitute the most heavyweight type of fault localisation

[1] For a repository of references of work about dynamic slicing in the context of fault localisation see [235].

methods in terms of computational cost [235], and are usually expected to have a high degree of effectiveness (which in practice comes at the expense of efficiency). In this section we overview major reasoning based approaches.

2.2.1 Program state-based approaches

A program state is a set of variable-value pairs for a given point in a given program execution. In general, program state based approaches propose to find faults by changing the values at various program states to see if the error disappears. Three prominent approaches are as follows:

1. *Delta debugging.* The strategy behind delta debugging is to remove useless program states that are not needed to reproduce the error. This is performed iteratively until a minimal program responsible for the error is found, and is performed using a divide and conquer method. Techniques that appeal to delta debugging include [49, 108, 175, 214–216, 247, 258].

2. *Predicate switching.* The strategy behind the predicate switching approach is to take a failing trace and switch the Boolean values of selected predicates [156, 229, 259] – if switching a given value results in a successful execution, then that predicate is "critical to the error", and that predicate is then proposed as a fault hypothesis. Automated methods such as SAT solvers are used to find a subset of predicates to switch, which can make the approach computationally costly.

3. *Value replacement* [123, 124]. The strategy behind the value replacement approach is to select values for a variable in a given failing execution and replace them with values from passing ones. If the replacement causes the error to disappear, then the original value is ranked as more suspicious.

2.2.2 Model-based approaches

In model-based fault localisation, it is not assumed that a correctly functioning or "golden model" of the program is available. Rather, models are constructed from the actual faulty program and tested as to whether it satisfies a given specification. Then, fault localisation information is constructed as a function of the model. Prominent model-based approaches include the following.

1. Using *Dependency-based* models. In general, these build a model of a program and use logical techniques to determine which parts of the model the error depends upon. Major approaches include Wotowa et al [244], Baah et al [30], and Mayer et al [168, 171]. Abstract interpretation methods have been introduced in an attempt to allay problems associated with efficiency [37, 52, 170].

2. *Model checking* approaches [31, 45, 101–103, 105, 106, 141]. In model checking approaches, a program is modelled in a formal language and the model is checked as to whether it satisfies a given specification or not. If it does not, then the program contains a fault, and a failing trace is produced as a "witness" for the error. In the simplest case, this trace can be investigated to help find the fault. Two prominent fault localisation approaches which build on this basic premise are as follows:

   (a) *Shortest failing trace approaches.* These use model checkers to find failing traces which cover the smallest amount of the program possible [85, 196, 205]. Shortest failing traces are useful, because given the premise there must be a fault covered by the failing trace, then the shorter the failing trace the less the user has to potentially look through in order to find a fault.

   (b) *Trace comparison approaches.* The *Explain* tool of Groce et al [103, 105, 106] uses a model checker to find two execution traces: The first is a failing trace, and the second is a passing trace which is most similar to the failing trace, where similarity is defined using a formally defined distance metric. The difference between the failing and passing trace provides a potential explanation for the error. The approach has also been used with slicing and predicate abstraction techniques [45]. A similar approach is the approach of Ball et al [31]. Here, model checkers are used to identify parts of the program which appear in a given failing trace but are not in the entire set of passing traces.

Three major problems with reasoning-based approaches are as follows: The first concerns effectiveness. A problem with shortest trace generation techniques is that it can often be the case that a large part of the program is covered, meaning a lot of the program will have to be investigated by the user. Similarly, a problem with the trace comparison techniques is that passing executions can be very different from failing traces, and thus there can be very few differences, leading to only a large area

localised as potentially faulty (see Griesmayer et al. [101] for discussion). This means that the techniques can suffer from poor effectiveness (measured in amount of code investigated by the user).

One attempt to increase the effectiveness of model checking based techniques, is to use them to make minimal changes to the faulty program in order to make failing traces not fail in re-executions. Those areas of minimal change are then presented as a hypothesis for the error. Different implementations include use of CBMC [101] and SATABS [102]. More recently, model checkers have been used in the context of automatically altering programs so that they produce fewer failing traces [60, 112, 136, 184].

The second problem is the general problem for all reasoning based techniques - that of efficiency. It has been observed that heavy use of model checkers make these approaches potentially time costly [102]. In general, reasoning and model based approaches have not been demonstrated to be able to scale to large programs (a few thousand lines or more, see [13] for discussion). Thus, more attention is now being given to more lightweight approaches (see [235] for additional discussion), where it is possible to leverage model checkers alongside these approaches.

Lastly, a problem with model checking techniques is that it is not always the case that a formal specification for the program can be found to be used with existing model checking methods. For example, in many software projects a golden program (perhaps a previous version which is assumed to be functionally correct) will exist, and failing and passing traces will be generated by comparing the outputs of both programs.

2.3 Spectrum-based Approaches

In this section we review work in *spectrum-based fault localisation* (SBFL). SBFL is currently one of the most prominent areas of software fault localisation research to date – estimated to consist of 35% of the literature (as measured by Wong et al.'s repository [235], who observe that "execution slice and program spectrum-based techniques have dominated since 2008."). SBFL's prominence is partly explained by the fact it is one of the most temporally efficient (or "lightweight") and scalable approaches. The advantage in efficiency stems from the fact that the approach does not require advanced modelling or reasoning techniques. Rather, fault-localisation information is determined purely as a statistical function of the coverage properties

of a given test suite. These test suites will often already pre-exist at the testing phase of the debugging cycle, meaning that fault localisation information can in practice be computed instantly – even for large (larger than 10k LOC) programs. This contrasts to reasoning-based approaches; which have reportedly only been shown to be able to scale to programs of more than a few hundred lines long [103, 170, 245, 252].

We outline the approach here. In general, SBFL takes the coverage information of execution traces of a test suite to determine how "suspicious" a given program component (such as block, predicate, executable statement) is with respect to being a fault. Here, a test suite consists of a set of test cases (executions) of the program. For each program component, a vector of statistics is determined (its program spectrum) as a function of the coverage details of the test suite. A suspiciousness measure is then defined as a function of this program spectrum, assigning a degree of suspiciousness to each program component. A list of components, ranked by suspiciousness, is then computed, and is used in one of either fault localisation paradigms, as follows:

1. *Semi-automated fault localisation.* In the conventional semi-automated paradigm, program components are manually inspected by the user in descending order of suspiciousness until a fault is found. This process has been demonstrated to help advanced engineers find faults in practice [188]. An early innovation in this paradigm was to use different colours to highlight each line of code with a colour proportional to its degree of suspiciousness (red/yellow/green for high/medium/low suspiciousness respectively). Tools such as ARISTOTLE integrate this feature into practical fault localisation tools [131, 132].

2. *Fully automated fault localisation.* In the (more recent) fully automated paradigm, approaches that inductively synthesise programs (such as CEGIS [125]) or repair programs (such as GENPROG [97]) have appealed to SBFL sub-routines to facilitate its fault localisation stage in efficient time frames. Here, the most suspicious component is identified as a function of a small test suite, and that component is targeted for automatic repair.

SBFL is described as a *ranking-based fault localisation method*, insofar as it ranks each program artefact by degree of suspiciousness. It therefore contrasts to (what we call) *categorical* fault localisation methods, which partitions program artefacts into two categories; those suspected to be faulty and those not.

To advance SBFL, work has concentrated in three major areas. First, finding the most effective suspiciousness measures (which measure the likelihood that a program

artefact is a fault as a function of its spectrum), second, defining spectra (the information which suspiciousness is designed to be a function of), and thirdly, improving the quality of test suites (in order to improve the quality of the spectra). We overview these areas of research as follows:

2.3.1 Developing measures

Finding the most effective suspiciousness measures are at the core of SBFL research. Good measures will rank faults with a relatively high degree of suspiciousness (meaning they be investigated early on by the user in the semi-automated paradigm), and bad measures will rank faults with a relatively low degree (meaning they will be inspected later). In this section we review the development of such measures.

The earliest studies of how to measure suspiciousness limit themselves to functions of failing traces (traces that violate the given specification) that a component is covered by [24, 144, 145, 218], but it is now consensus that this is ineffective and that passing traces (traces that satisfy the given specification) also help the fault localisation process [23, 71, 130]. Following these, early techniques include *set union*, *set difference*, and *nearest neighbour* techniques [197], and demonstrated improvement in experimentation. The first identifies a suspicious set with the components covered by a failing trace but which are not covered by the union of the passing traces, the second with components covered by a failing trace but which are not covered by the intersection of the passing, and the third with components covered by a failed trace but are not covered by a single most similar passing trace.

Since these techniques, research in SBFL has largely been driven by the introduction, motivation, and experimental comparison of new suspiciousness measures [18, 19, 40, 71, 130, 140, 149, 150, 158, 161–163, 181, 194, 197, 238, 241, 242, 246, 254, 262]. The established convention is that suspiciousness measures map a program artefact to a real number (degree of suspiciousness) as a function of its program spectrum $\langle a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np} \rangle$, where a^i_{ef} is the number of failing traces that cover the i-th program component, a^i_{ep} is the number of passing traces that cover the i-th program component, a^i_{nf} is the number of failing traces that do not cover the i-th program component, and a^i_{np} is the number of passing traces that do not cover that component. The higher/lower the degree of suspiciousness the more/less suspicious the program component C_i is assumed to be with respect to being a fault. In what follows, we drop the index i when the context is clear. Currently, there are (to our knowledge) at least 110 measures in the SBFL literature. This includes 32 similarity measures selected by Naish [181], the 40 association measures of Lo [163], the 30 genetically

learned measures of Yoo [254], the 2 Dstar measures [242] D2 and D3, and the 6 "combination" measures of Kim et al [140]. Tables providing the formal definitions of large classes of measures are given in Appendix A.1 for the interested reader.

We now present a broad outline of the different approaches that developers of suspiciousness measures have taken. Methods and motivation for different measures have been many and varied. In general, measures have either been developed *manually* (created by hand), or *automatically* (by machine), and motivation has either involved theoretical (*a priori*) motivation – insofar as it has been developed to satisfy certain intuitions or formal properties about fault localisation, or empirical (*a posteriori*) motivation – insofar as it has been developed purely to perform well on a set of benchmarks.

We first discuss the manually-created measures. The first group of measures used in SBFL were hand-designed for the specific purposes of fault localisation. Two well known measures are Tarantula [133] and Wong-III [241]. Tarantula was the first measure to be defined in terms of a program spectrum (as defined above), and was developed around the intuition that the more failing traces and fewer passing traces that cover a component, the more suspicious it is [130]. Wong-III was developed around the observation that that the weight of passing traces should change according to the number of passing traces, as follows:

$$Tarantula = \frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}}$$

$$Wong\text{-III} = a_{ef} - h, \text{ where } \begin{cases} h = a_{ep} & \text{if } a_{ep} \leq 2 \\ h = 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ h = 2.8 + 0.01(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$$

A second group of manually-created measures are distinctive insofar as they were not customised to SBFL (and so are unlike Tarantula and Wong-III above). Instead, they were originally designed and motivated for application in other domains, and were put to novel use in the domain of SBFL. This included many similarity measures and bio-metrics which did not have a direct connection to the SBFL problem. Many of these measures were presented by Naish [181] and Lucia [163]. Two prominent examples are Jaccard [121] and Ochiai [185]. The Jaccard measure is a general similarity measure [121], and the Ochiai measure was originally developed in early 20th century Japan for the categorisation of fish [185]. Despite not being directly connected to

SBFL, many such measures have been consistently shown to perform well in SBFL experimentation [18]. Prominent similarity measures are as follows:

$$Ochiai = \frac{a_{ef}^2}{(a_{ef} + a_{nf})(a_{ef} + a_{ep})}$$

$$Jaccard = \frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep}}$$

$$Kulkzynski = \frac{a_{ef}}{a_{ep} + a_{nf}}$$

A third group of manually created measures were developed according to a strategy of taking a pre-existent measure taken from other domains, and then manually tweaking them in order to improve experimental performance. Two prominent examples are the Zoltar [89] and Dstar [242] measures, which were developed by developing the Jaccard and Kulkzynski measures (described above) respectively, as follows.

$$Zoltar = \frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep} + \frac{10000 a_{nf} a_{ep}}{a_{ef}}}$$

$$D2 = \frac{a_{ef}^2}{a_{ep} + a_{nf}}$$

To see the connection between Jaccard and Zoltar, we observe that they are the same measure except that $\frac{10000 a_{nf} a_{ep}}{a_{ef}}$ has been added to the sum of Jaccard's denominator. To see the connection between Kulkzynski and D2, we observe that they are the same measure except the power of Kulkzynski's numerator has been raised to a_{ef}^2 (D3 raises it to a_{ef}^3). In both these cases, the general effect is to make the number of failing executions count a lot more with respect to suspiciousness than passing traces. The motivation for this comes empirically - measures which weigh failing traces more heavily than passing traces have experimentally performed better [242].

A fourth group of manually created measures that were designed specifically for the purposes of fault localisation were ones which aimed to solve certain sub-problems in SBFL. Namely, these measures were developed to perform optimally when there is only a single faulty component in the program. The reasoning is that if there is only a single fault in the program, then all failing traces must cover the faulty component, in which case components covered by all failing traces should be most suspicious. One measure of this sort is the Naish measure [181], as follows:

$$Naish = a_{ef} - \frac{a_{ep}}{a_{ep} + a_{np} + 1}$$

Here, the condition is that components covered by more failing traces are more suspicious, and passing traces only count as tie breakers between components covered by the same number of failing traces. Accordingly, the measure will make components covered by all failing traces (including a fault in a single fault program) as most suspicious.

Finally, we discuss automatically generated measures. Yoo [254] used automated methods to generate suspiciousness measures. Here, a set of faulty programs were used as a training set to develop suspiciousness measures using genetic algorithms. Many measures developed were complex, but in general confirmed the view that the number of failing traces was more important than the number of passing traces in establishing suspiciousness. An example of an effective measure generated by appeal to genetic algorithms is as follows [254]:

$$G26 = 2a_{ef}^2 + \sqrt{a_{np}}$$

We now discuss theoretical results about measures. Theoretical results include proving desirable formal properties of measures and finding equivalence proofs for classes of measures [58, 149, 179, 181, 246]. Discussed properties include; *strict rationality* – which states that suspiciousness should strictly increase on the number of failing traces and strictly decrease on passing traces [179]; *single bug optimality* – which states that program components covered by all failing traces are the most suspicious [181]; *equivalence* – which states when two suspiciousness measures are equivalent for the purposes of ranking program components [58, 149, 181]. Xie et al also prove optimality results for measures on programs with a single fault [246]. We discuss these properties formally and in greater detail in Chapter 5.

Finally, there is additional work on what to do when two program components are ranked as equally suspicious by a measure. Tie-breaking techniques include taking statement execution frequency and or/order into account [14, 151]. There are also confidence-based, and data dependency-based approaches [191].

We now discuss two outstanding problems with current research. The first is a theoretical problem – thus far, there have not been many formal properties about the general problem of fault localisation which SBFL measures have been shown to satisfy. Although there has been some discussion of a few properties that measures should

satisfy a priori (such as strict rationality [179]), and measures that solve fault localisation sub-problems have been presented (such as single bug optimal measures [181]), there is not yet a SBFL measure that solves the problem of fault localisation for all benchmarks. Indeed, recently Yoo et al. have established theoretical results which show that a "best" performing suspicious measure does not exist [253]. In light of this, the SBFL literature has favoured developing measures with good experimental performance as opposed to developing them according to a priori requirements. As Wong writes, "because it is not possible to theoretically prove if one fault localization technique shall always be more effective than another, such empirical validation is typically the norm for fault localization studies" [242]. The norm of finding the best measure experimentally has facilitated a culture of borrowing measures from other domains [149,163,181], manually tweaking measures [241,242], or using machine learning methods [238,249,254]. Thus, if SBFL cannot be shown to satisfy many properties key to fault localisation, there then remains the project of developing equally lightweight frameworks which can.

The second is an experimental problem. To compound the problem that there has not yet been found a best measure in theory, there has also not been found a measure which performs best in practice – clearly and consistently outperforming all other measures. This was the conclusion of Lucia et al. who performed a large scale comparison of 40 association measures on programs ranging from 1 to 5 faults [165]. A related experimental problem is that all experiments thus far have been quite small - usually performed on a small number of small benchmarks from the SIR testing repository [7], over programs with a range of 1 to 5 faults [13, 165, 181]. There is thus the challenge of providing much larger experimentation over larger programs, with a greater range of faults, comparing a greater quantity of measures, and finding measures which perform best overall.

In summary, outstanding issues include finding new efficient methods which satisfy theoretical properties key to fault localisation, and which perform competitively in large scale experimentation.

2.3.2 Defining spectra

We now discuss the different types of spectra. Work has been done as to what sorts of information are useful to include in a program spectrum in order to gauge suspiciousness, and has usually at least involved the number of passing/failing traces that do/do not cover a given component. We discuss four additional major approaches.

1. First, there are approaches which leverage invariants to define spectra. These are known as *program invariants hit spectrum* (PIHS) approaches [235]. In general, invariants are properties which are always true for a program (under certain conditions). It is accordingly assumed that knowledge of what is invariant to failing traces is valuable in the project of fault localisation. There are both approaches which use *strict invariants* [72], and a looser criterion of *potential invariants* [195] or *likely invariants* [203]. In addition *extended invariants* have been defined which adds execution counts to invariants [27].

2. Second, there are *predicate count spectrum based* approaches (PCRS) [235], which counts how many times predicates are executed. Approaches include the approach of Liblet [158], SOBER [161], and have been shown to be effective [178].

3. Third, there are *method calls sequence hit spectrum* (MCSHS) approaches [235]. These take into account the execution sequence of method calls [54, 169].

4. Fourth, there are *time spectrum* approaches, which record the execution time of every method in passing/failing executions, the differences being analysed for indications of faultiness [251].

Additional approaches which add information to a spectrum are as follows. First, there is the Crosstab method [239] which computes suspiciousness by appeal to cross-tabulations of covered/not covered categories against passing/failing executions. Zhang et al. [261] identify "short-circuit" evaluations of predicates to aid in fault localisation. You et al. [256] propose a statistical approach that analyses behaviours of sequentially connected predicates. Modi et al. explore execution phases such as CPU and memory usage [176].

In summary, common to the vast majority of definitions of spectra is the use of four pieces of information: the number of passing/failing traces that do/do not cover a given component; however leveraging additional statistics about the program can potentially improve fault localisation effectiveness.

2.3.3 Improving test suites

In this section we discuss methods which aim to improve test suites usable for SBFL. Test suites form the foundational dataset which spectra are based on, and are thus fundamental to the effectiveness of the SBFL approach. In SBFL it is usually

assumed that a large test suite with a set of identified passing/failing test cases is available to the engineer and is of sufficient quality for fault localisation. However, this may not be the case in practice. As a consequence there is work being done on improving the quality of test suites. The major areas of research are as follows:

1. *Generating test case oracles.* First, there is the problem of finding oracles for test suites, where an oracle determines whether a given execution is passing or failing. In real world scenarios, such oracles may not always be available – studies suggest that many contexts in industry do not have automated oracles otherwise facilitated by the existence of formal specifications or assertions [115, 116]. Furthermore hand-written oracles may be "incomplete, out-of-date, or ambiguous" [235]. Accordingly, Jahangirova et al. develop a technique to assess and improve the accuracy of test oracles [122].

2. *Test suite expansion.* Second, there is the problem of improving the test suite by the addition of further test cases. Here it is generally assumed we have a testing oracle. A first criterion is that the test suite has sufficient coverage of the program. To this end, Diaz et al. develop their *Tabu Search*, which is an automatic method which uses meta-heuristics to generate test suites with maximum coverage [62]. Artzi et al. developed their *Apollo* tool, which is an automatic method to generate test suites based on concrete and symbolic executions [29]. Studies suggest that test suites with high branch coverage are better than high statement coverage for the purposes of fault localisation [127]. In general, it has been shown that improving different types of coverage also improves fault localisation for SBFL [204]. Feldt et al. study how different types of coverage affects fault localisation [76], and conclude that test suites with larger test set diameter (as measured by their measure (I-TSDm)) potentially have better fault-finding ability. Aside from improving coverage, the BUGEX tool is a method which generates test cases with a minimal distance from a given failed trace [200], and a similar approach is applied for SBFL [128]. Baudry et al. use a bacteriological approach in order to generate test suites which are simultaneously facilitate both testing and fault localisation [34]. Finally, concolic execution methods (which integrate both concrete and symbolic executions) have been developed to incrementally add test cases to a test suite based on their similarity to an initial failing run [29, 206].

3. *Test suite reduction.* Recently, many approaches have demonstrated that it is not necessary for all test cases to be used. Rather, one can use *test suite reduction* (which selects a handful of test cases) in order to minimise the number of test cases required for fault localisation [41, 42, 88, 90, 111, 173, 226, 226, 250, 257]. Most approaches are based on a strategy of eliminating redundant test cases relative to some coverage criterion. The effectiveness of applying various coverage criteria in test suite reduction is traditionally based on empirical comparison of two metrics: one which measures the size of the reduction, and the other which measures how much fault detection is preserved. Studies of the effectiveness of different reduction techniques is given in Gonzalez et al [91].

4. *Test case prioritisation.* Test case prioritisation methods weigh different test cases differently according to priority [32, 92, 126]. Coverage based approaches include those which weigh failing test cases which cover fewer statements more [180], and weigh failing test cases which duplicate the coverage details of other failing test cases less [84]. Information theoretic approaches include the FLINT technique which prioritises test cases according to an information theoretic measure [255].

5. *Slicing.* A prominent approach to improving the quality of test suites involves the process of slicing test cases. Here, SBFL proceeds as usual except the program and/or the traces composing the test suite are sliced [28, 118, 152, 233]. For example, Alves et al. [28] combine Tarantula along with dynamic slices, Ju et al. [135] use SBFL in combination with both dynamic and execution slices in their HSFal (hybrid slice spectrum fault locator) tool, and a third approach extends the combination of SBFL and slicing even further by using automated methods (to generate hitting sets) [118] in the HS-Slice algorithm [243]).

6. *Model checking based techniques.* Other techniques designed to improve the test suites for fault localisation include ones which use model checkers [96]. Gopinath et al. propose using SAT solvers to find minimal unsatisfiable cores within a failing trace, where this core consists of a minimal unsatisfiable set of variables in the program, and that these cores be added as failing traces to a test suite. This adds more weight to parts of the code which correspond to such cores.

2.4 Multiple-Technique Combination

Wong et al. observe that "the effectiveness of a fault localization technique is very much scenario-dependent, affected by successful and failed test cases, program structures and semantics, nature of the bugs, etc. There is no single technique superior to all others in every scenario. Thus, it makes sense to combine multiple techniques" [235]. Accordingly, there have been attempts to combine lightweight SBFL methods with more heavyweight techniques. We observe that these attempts follow one of two different strategies:

1. *Series approach.* The first approach is to use light and heavyweight techniques in *series*. The idea is to first use a lightweight approach (such as SBFL) in order to get general fault-localisation information, and secondly refine this information using a more heavyweight technique. This approach was implemented by the DEPUTO tool of Abreu et al. [15], who demonstrated that lightweight and heavyweight methods can be used successfully to complement one another.

2. *Parallel approach.* A second approach is to use different fault-localisation techniques in *parallel*. The idea is to generate new fault-localisation information as a combined function of different SBFL measures. This approach has been implemented by Debroy et al. [56,59], who propose a method to combine the fault localisation rankings of multiple techniques in a "consensus based" approach. Similarly Lucia et al. develop a "fusion fault-localisation" approach which combines results of multiple SBFL measures after normalising their results [165]. Similarly, Wang [228] use simulated annealing and genetic algorithms to find good combinations of different measures. Additionally, Xuan et al. consider simple combinations of measures to make new measures [140, 249].

In general, different techniques do not need to be considered as rivals competing against each other. Rather, the approaches can used in combined approaches to advance fault localisation.

2.5 Multiple-Fault Localisation Approaches

Thus far we have reviewed techniques which are designed to find a *single* fault in a program (i.e. perform single fault localisation). However, real world programs usually contain *multiple* faults. Studies report the industry average is "about $12-50$ errors

per 1000 lines of delivered code", and similarly Microsoft applications have "about 10−20 defects per 1000 lines of code during in-house testing" [172]. Additional studies show that individual failures are often caused by a complex of multiple faults that are spread across the program [110,166]. Thus, if real programs typically contain multiple faults, there remains the problem of developing techniques to localise multiple faults. Multiple fault localisation (MFL) is a recent, but growing, area of research [13,16,17, 20,21,44,55,63,87,124,129,160,164,191,211,212,230,263,264]. We observe there are two major strategies which have been employed to perform MFL, and are called in the literature a *one-at-a-time* strategy, and an *all-at-once* strategy [235].

1. *One-at-a-time strategy.* This strategy is implemented as follows: after a single fault is found using a given technique, the fault is repaired, and another independent round of single fault localisation is performed. If an SBFL technique is used, this may require that the test suite is re-executed or re-generated. A major problem with this approach is its efficiency: multiple re-executions/generations of test suites can be undesirable as it can often take a prohibitively long time to do so.

2. *All-at-once strategy.* An alternative and potentially time saving strategy is to find multiple faults in one sitting, resisting the requirement of the one-at-a-time strategy of re-executing after a single fault is found. A major problem with this approach is its effectiveness. This is because finding an effective all-at-once method presents a major challenge as it is harder than finding a single fault in a program – by definition the former is a sub-problem of the latter. Furthermore, studies suggest that multiple-fault localisation is a much more complex problem in practice given the complex interference patterns of multiple faults [57].

Currently, the incentive in the literature is to improve on the effectiveness of all-at-once approaches. Prominent coverage-based approaches have been based on the observation that the set of faults must be covered by all the failing traces (i.e. it must be "a hitting set"), and thus we can use algorithms to find sets of program components which satisfy this property to serve as hypotheses for the error [13, 16, 16, 17, 20, 21, 55, 211]. Three approaches that use such algorithms are as follows. First, the most prominent tool is the Bayesian approach of BARINEL, which uses a Bayesian method to rank hypotheses according to estimated likelihood [13, 16, 17, 20, 21], and a customised tool STACCATO to approximate the set of minimal hitting sets [16]. Secondly, there is the frequentist approach of Steimann et al, who propose

a technique in which the program component which is in the *most* hitting sets is the most suspicious [211]. Their approach differs from Abreu's insofar as it does not use passing traces, but is in general intractable other than for small problems as it requires all hitting sets. Third, a less prominent technique that only requires one hitting set is the approach of Dean [55], in which linear programming is used to approximate the problem of finding a hitting set with a small number of passing traces covering it. In general, approaches which use automated reasoning techniques to find hitting sets have been shown to come at a time cost, and have not yet been demonstrated to scale to very large programs.

We now turn to attempts to use SBFL techniques for MFL. Gong et al. [87] propose continuing any SBFL process if the faults found thus far are not collectively covered by all failing traces – since there must be further undiscovered faults in the program to be responsible for the remaining traces. Lucia et al. [164] experimentally investigate using SBFL techniques for locating all faults in an all-at-once strategy. They conclude with pessimistic results, showing that usually most of the program needs to be investigated in order to find all faults on a range of 2-5 fault programs. They conclude that "the accuracies of the measures in localizing multi-bug programs are lower than single-bug programs, which provokes future research" [163]. Reasons for why established SBFL methods have failed at MFL has been investigated, and is explained to increased noise created by the introduction of multiple faults [63].

Past methods to overcome the problems facing SBFL at MFL have largely focused on using *clustering* methods on test suites, which involves partitioning the test suite into multiple smaller test suites in order to simplify a large multiple fault problem into smaller problems [129, 160, 191, 212, 230, 263, 264]. Jones et al. propose clustering failed executions and distributing each cluster to different debuggers [129] in a parallel debugging method. Other approaches are as follows: Zheng et al. [263] propose a clustering method in order to group failed executions (identifying one feature that characterizes each cluster), Steimann et al. [212] use integer linear programming to create clusters, Wei proposes a clustering technique based on correlation coefficients [230], Zheng proposes a predicate based clustering approach [264], Lucia et al. propose a clustering approach for simulink models [160], and lastly Podgurski introduces a pattern classification and multivariate visualisation technique to form clusters [191].

The main problem with clustering techniques is that a large number of faults can potentially remain in each cluster/test suite [212]. Consequently, even though clustering techniques have been demonstrated to help SBFL find multiple faults more

effectively, there still remains original problem of developing SBFL techniques which can find multiple faults in the given test suite.

Miscellaneous multiple fault localisation techniques include Ducasse et al.'s data mining technique [44], which appeals to formal concept analysis and association rules, but is demonstrated only on small examples. Jeffrey's value replacement method has also been attempted for multiple faults, but has high overhead [124].

In summary, attempts have been made to develop effective multiple fault localisation techniques. Mid-weight techniques such as BARINEL demonstrate the plausibility of MFL on smaller programs [13, 20, 21]. However, SBFL approaches have been demonstrated to be poor at locating multiple faults using established search methods [21, 163], and clustering techniques still create partitioned test suites which still contain large numbers of faults. Thus, there remains the problem of developing techniques which can substantially improve SBFL at multiple fault localisation on test suites with multiple faults. Effective methods of this sort would be expected to perform well in experiments without appeal to clustering.

2.6 Research Directions

In terms of research directions which can help develop the most effective and efficient fault-localisation techniques, we conclude that research into lightweight statistical techniques comparable to SBFL provide for the most promising directions – in particular, with respect to both the problems of single and multiple fault localisation. Major research directions suggested by our literature review are summarised as follows:

1. Given the established efficiency of SBFL, a first major research direction involves attempts to advance the existing SBFL framework – which currently lies at the state-of-the-art in lightweight statistical fault localisation. A prominent research priority is to find new and most effective SBFL measures. *Desiderata* include first showing that these measures are sufficiently well motivated to SBFL, and second showing that the new measures substantially and significantly outperform (or are at least competitive with) established measures at fault-localisation tasks in terms of effectiveness on larger scale experimentation.

2. At single fault localisation, SBFL has various limitations. Recent theoretical results have formally established that no "best" SBFL measure can be developed

to solve the problem of fault localisation [253], and recent experimental results have established that no measure performs "best" overall [164], thus it is an open question as to whether a similar and equally lightweight method can be developed to be an improvement on SBFL. *Desiderata* for this method includes firstly showing that it satisfies and exploits desirable formal properties key to fault localisation (which SBFL does not), and second showing that using the new method substantially and significantly outperforms SBFL at a range of fault localisation tasks.

3. At multiple fault localisation (MFL), lightweight techniques (such as SBFL) similarly have limitations. Recent work has experimentally established that SBFL methods are poor at multiple fault localisation [164], and despite the advent of clustering methods, there still remains the problem of adapting SBFL to test suites containing large numbers of faults. Thus, a third major research direction is to develop such techniques. *Desiderata* include firstly showing that the technique satisfies and exploits desirable formal properties key to MFL which others do not, and secondly showing that the technique substantially and significantly improves on ranking based methods (such as SBFL, and BARINEL) at a range of MFL tasks.

4. Generating test suites for use with ranking based fault localisation methods, such as SBFL, is important for improving effectiveness. Accordingly, a prominent research direction is to identify how to efficiently optimise existing test suites (or generate small test suites from scratch) which satisfy formal properties exploitable by given techniques.

The thesis contributions presented in Section 1.4 are designed to advance these research directions.

Chapter 3

Preliminaries

In this chapter the preliminaries common to the theories discussed in chapters 5, 6, 7, and 8 are presented.

3.1 Probands

Using the terminology of Steimann et al. [213], a *proband* is a faulty program together with its test suite that we use for evaluating the performance of a given fault localization method. In the remainder of this section we describe faulty programs and test suites in more detail.

First, a *faulty program* is a program which fails to always satisfy a specification, where a specification is a property expressible in some formal language and describes the intended behaviour of some part of the program under test. When a specification fails to be satisfied for a given execution (i.e. an error occurs), it is assumed there exists something in the program which was the cause of that error, identified as the fault for that execution.

Example 3.1.1. An example of a faulty C program is given in Figure 3.1 (called `minmax.c`). This example is taken from Groce et al. [103], and we shall use it as our running example throughout this chapter. There are some executions of the program in which the assertion `least <= most` is violated, and thus the program fails to always satisfy the specification. In such executions, the cause of the error is the fault marked `C3`, which should be an assignment to `least` instead of `most`.

Second, a *test suite* is a collection of test cases whose result is independent of the order of their execution, where a *test case* is an execution of some part of a program. Each test case is associated with an input vector, where the nth value of the vector is

```
int main() {

  int input1, input2, input3;
  int least = input1;
  int most = input1;

  if (most < input2)
    most = input2;  // C1

  if (most < input3)
    most = input3; // C2

  if (least > input2)
    most = input2; // C3 (fault)

  if (least > input3)
    least = input3; // C4

  assert(least <= most); // E
  }
```

Figure 3.1: `minmax.c`

assigned to the nth input of the given program for the purposes of a test (according to some given method of assigning values in the vector to inputs in the program). Each test suite is associated with a set of input vectors which can be used to generate the test cases. A test case *fails* (or is *failing*) if it violates a given specification, and *passes* (or is *passing*) otherwise.

Example 3.1.2. We give an example of a test case for the running example. The test case with associated input vector $\langle 0, 1, 2\rangle$ is an execution in which `input1` is assigned 0, `input2` is assigned 1, and `input3` is assigned 2, the statements labelled `C1` and `C2` are executed, but `C3` and `C4` are not executed, and the assertion is not violated at termination, as `least` and `most` assume values of 0 and 2 respectively. Accordingly, we may associate a collection of test cases (a test suite) with a set of input vectors. For the running example the following ten input vectors are associated with a test suite of ten test cases: $\langle 1, 0, 2\rangle$, $\langle 2, 0, 1\rangle$, $\langle 2, 0, 2\rangle$, $\langle 0, 1, 0\rangle$, $\langle 0, 0, 1\rangle$, $\langle 1, 1, 0\rangle$, $\langle 2, 0, 0\rangle$, $\langle 2, 2, 2\rangle$, $\langle 1, 2, 0\rangle$, and $\langle 0, 1, 2\rangle$. Here, the first three input vectors result in error (and thus their associated test cases are failing), and the last seven do not (and thus their associated test cases are passing).

Thirdly, an important concept in fault localisation is a *unit under test* (UUT). A UUT is a concrete artefact in a program which is a candidate for being at fault. Many types of UUTS have been defined and used in the literature, including classes [54],

methods [212], blocks [18,63], branches [204], and statements [131,158,240]. A UUT is said to be *covered* by a test case just in case that test case executes the UUT. For simplicity and convenience, it will help to always think of UUTS as being labelled `C1`, `C2`, `C3`, ... etc. in the program itself. Assertion statements are not considered to be UUTs.

Example 3.1.3. To illustrate some UUTs for the running example (Figure 3.1), we have chosen the units under test to be the statements labelled in comments marked `C1`, `C2`, `C3`, and `C4`. The assertion is labelled `E`, which is violated when an error occurs. To illustrate a proband, the faulty program `minmax.c` (described in example 3.1.1), and the test suite associated with the input vectors described in example 3.1.2, together describe a proband.

Finally, we present some assumptions about probands. It is assumed that test suites satisfy the following two assumptions [213]: Every failing test case covers at least one UUT, and every faulty UUT is covered by at least one failing test case. The user is assumed to be an oracle who can correctly determine, upon inspection of the faulty program, which UUTs are faulty and which are not. In an experimental setting the faulty components are always known before inspecting the program (a premise which makes experimental evaluation possible), whereas in practice they are not.

3.2 Proband Models

In this section we define proband models, which are the principle formal objects used by the fault localization techniques discussed in this thesis. Informally, a proband model is a mathematical abstraction of a proband. In our development we assume the existence of a given proband in which the UUTs have already been identified for the faulty program and appropriately labelled `C1`, ..., `Cn`, and assume a total of n UUTs. We begin as follows.

Definition 3.2.1. A set of *coverage vectors*, symbolised $\mathbf{T}$, is a set $\{t_1, \ldots, t_{|\mathbf{T}|}\}$ in which each $t_k \in \mathbf{T}$ is a coverage vector defined $t_k = \langle c_1^k, \ldots, c_{n+1}^k, k\rangle$, where

- For all $0 < i \leqslant n$, $c_i^k = 1$ if the ith UUT is covered by the test case associated with t_k, and 0 otherwise.
- $c_{n+1}^k = 1$ if the test case associated with t_k fails and 0 if it passes.

Intuitively, each coverage vector can be thought of as a mathematical abstraction of an associated test case which describes which UUTs were executed in that test case. We will also use the following additional terminology and notation. If the last argument of a coverage vector in $\mathbf{T}$ is the number k it is symbolised t_k. k is in the range $0 < k \leq |\mathbf{T}|$ and uniquely identities a coverage vector in $\mathbf{T}$. t_k is also called the kth coverage vector and is said to correspond to the kth test case in a test suite. In general, for each $t_k \in \mathbf{T}$, c_i^k is the value of the ith argument in t_k. If $c_{n+1}^k = 1$ then t_k is also described as a *failing* coverage vector, and as *passing* otherwise. The set of failing/passing coverage vectors is denoted $\mathbf{F}/\mathbf{P}$ respectively. c_{n+1}^k is also denoted e^k (as it describes whether the error occurred). Coverage vectors are also called *traces*. For convenience, we may represent the set of coverage vectors $\mathbf{T}$ with a *coverage matrix*, where for all $0 < i \leqslant n$ and $0 < k \leqslant |\mathbf{T}|$ the cell intersecting the ith column and kth row is c_i^k and represents whether the i-th UUT was covered in the test case corresponding to t_k. The cell intersecting the last column and kth row is e^k and represents whether t_k is a failing or passing trace.

Example 3.2.1. For the test suite described in 3.1.2 we can describe a set of coverage vectors $\mathbf{T} = \{t_1, \ldots, t_{10}\}$ in which $t_1 = \langle 0, 1, 1, 0, 1, 1\rangle$, $t_2 = \langle 0, 0, 1, 1, 1, 2\rangle$, $t_3 = \langle 0, 0, 1, 0, 1, 3\rangle$, $t_4 = \langle 1, 0, 0, 0, 0, 4\rangle$, $t_5 = \langle 1, 0, 0, 0, 0, 5\rangle$, $t_6 = \langle 0, 0, 0, 1, 0, 6\rangle$, $t_7 = \langle 0, 0, 1, 1, 0, 7\rangle$, $t_8 = \langle 0, 0, 0, 0, 0, 8\rangle$, $t_9 = \langle 1, 0, 0, 1, 0, 9\rangle$, and $t_{10} = \langle 1, 1, 0, 0, 0, 10\rangle$. Here, coverage vector t_k is associated with the kth input vector described in the list in Example 3.1.2. To illustrate how input and coverage vectors relate, we observe that t_{10} is associated with a test case with input vector $\langle 0, 1, 2\rangle$ which executes the statements labelled `C1` and `C2`, does not execute the statements labelled `C3` and `C4`, and does not result in error. Consequently $c_1^{10} = c_2^{10} = 1$, and $c_3^{10} = c_4^{10} = e^{10} = 0$, and $k = 10$, such that $t_{10} = \langle 1, 1, 0, 0, 0, 10\rangle$ (by the definition of coverage vectors). A coverage matrix representing $\mathbf{T}$ is given in Table 3.1. In practice, given a program and an input vector, one can extract coverage information from the associated test case by using established tools (for example, for C programs, Gcov [3] can be used).

Definition 3.2.2. Let $\mathbf{T}$ be non-empty set of coverage vectors, then $\mathbf{T}$'s *program model* $\boldsymbol{PM}$ is defined as an ordered set $\langle C_1, \ldots, C_n\rangle$, where for each $C_i \in \boldsymbol{PM}$, $C_i = \{t_k \in \mathbf{T} | c_i^k = 1\}$.

We shall often use the notation $\boldsymbol{PM}_{\mathbf{T}}$ to denote the program model $\boldsymbol{PM}$ associated with $\mathbf{T}$. $C_{|\boldsymbol{PM}|}$ is also denoted E (denoting the event of the error). Each member of a program model is called a *program component* or *event*, and if $c_i^k = 1$ we say C_i *occurred* in t_k, that t_k *covers* C_i, and say that C_i is *faulty* if its corresponding UUT

	C_1	C_2	C_3	C_4	E
t_1	0	1	1	0	1
t_2	0	0	1	1	1
t_3	0	0	1	0	1
t_4	1	0	0	0	0
t_5	0	1	0	0	0
t_6	0	0	0	1	0
t_7	0	0	1	1	0
t_8	0	0	0	0	0
t_9	1	0	0	1	0
t_{10}	1	1	0	0	0

Table 3.1: coverage matrix

is faulty. Following this, each event can be intuitively thought of as the set of vectors in which it occurs.

Example 3.2.2. We use the running example to illustrate an example program model. For the set of coverage vectors $\mathbf{T} = \{t_1, \ldots, t_{10}\}$, we may define a program model $\boldsymbol{PM} = \langle C_1, C_2, C_3, C_4, E\rangle$, where $C_1 = \{t_4, t_9, t_{10}\}$, $C_2 = \{t_1, t_5, t_{10}\}$, $C_3 = \{t_1, t_2, t_3, t_7\}$, $C_4 = \{t_2, t_6, t_7, t_9\}$, $E = \{t_1, t_2, t_3\}$. Here, we may think of $C_1, \ldots, C_4$ as events which occur just in case a corresponding UUT (lines of code labelled `C1, C2, C3, C4` respectively) is executed, and E as an event which occurs just in case the assertion `least <= most` is violated. C_3 is identified as the faulty component.

Definition 3.2.3. For a given proband we define a *proband model* $\langle \boldsymbol{PM}, \mathbf{T}\rangle$, consisting of the faulty program's program model $\boldsymbol{PM}$, and the test suite's set of coverage vectors $\mathbf{T}$.

For example, the program model $\boldsymbol{PM}$ and set of coverage vectors $\mathbf{T}$ described in the running example together describe a proband model $\langle \boldsymbol{PM}, \mathbf{T}\rangle$. Finally, a definition which will be used in chapter 7 is as follows.

Definition 3.2.4. *Faulty hitting sets.* Let $\langle \boldsymbol{PM}, \mathbf{T}\rangle$ be a proband model where $C_i, \ldots, C_j \in \boldsymbol{PM}$ and we let $X = C_i \cup \cdots \cup C_j$. Then, for the set of failing coverage vectors $\mathbf{F} \subseteq \mathbf{T}$, if $\mathbf{F} \subseteq X$ then X is a *hitting set on* $\mathbf{F}$. Furthermore, if each of $C_i, \ldots, C_j \in \boldsymbol{PM}$ are faulty, then we say X is a *faulty hitting set.*

As is established [13], it is assumed that the set of faults in a faulty program will correspond to a faulty hitting set.

Example 3.2.3. We give an example of a faulty hitting set in the running example. We observe that $\mathbf{F} = \{t_1, t_2, t_3\}$, and $C_3 = \{t_1, t_2, t_3, t_7\}$. Thus $\mathbf{F} \subset C_3$, and so C_3 is a hitting set on $\mathbf{F}$ according to the definition. Furthermore, as C_3 is faulty, C_3 is also a faulty hitting set. Note that the faulty hitting set consists of one single program component in this example. In general, in programs with multiple faults they can be much larger.

3.3 Program Spectra

In this section we define program spectra, which is a principle formal object used in spectrum based fault localization.

Definition 3.3.1. For each proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$, and each $C_i \in \boldsymbol{PM}$, a component's *program spectrum*, is a vector $\langle a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np} \rangle$, where

- $a^i_{ef} = |\{t_k \in \mathbf{T} | c^k_i = 1 \wedge e^k = 1\}|$
- $a^i_{nf} = |\{t_k \in \mathbf{T} | c^k_i = 0 \wedge e^k = 1\}|$
- $a^i_{ep} = |\{t_k \in \mathbf{T} | c^k_i = 1 \wedge e^k = 0\}|$
- $a^i_{np} = |\{t_k \in \mathbf{T} | c^k_i = 0 \wedge e^k = 0\}|$

Informally, a^i_{ef} is the number of failing coverage vectors in $\mathbf{T}$ that cover C_i, a^i_{nf} is the number of failing coverage vectors in $\mathbf{T}$ that do not cover C_i, a^i_{ep} is the number of passing coverage vectors in $\mathbf{T}$ that cover C_i, and a^i_{np} is the number of passing coverage vectors in $\mathbf{T}$ that do not cover C_i. We often drop the numerical indices when the context is clear, writing a_{ef} instead of a^i_{ef}, and C instead of C_i etc. We observe that a^i_{ef}, a^i_{nf}, a^i_{ep}, and a^i_{np} are equal to $|C_i \cap E|, |\overline{C_i} \cap E|, |C_i \cap \overline{E}|$ and $|\overline{C_i} \cap \overline{E}|$ respectively.

Example 3.3.1. For the proband model of the running example $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ (where $\boldsymbol{PM} = \langle C_1, \ldots, E \rangle$ and $\mathbf{T}$ is represented by the coverage matrix in Table 3.1), the spectra for C_1, C_2, C_3, C_4, and E are $\langle 0, 3, 3, 4 \rangle$, $\langle 1, 2, 2, 5 \rangle$, $\langle 3, 0, 1, 6 \rangle$, $\langle 1, 2, 3, 4 \rangle$, and $\langle 3, 0, 0, 7 \rangle$ respectively.

We may represent each component's program spectrum on a 2×2 *contingency table* [190], where the number in a cell represents the frequency of the intersection of the events at that cell's respective cell and column (thus, the number in the cell at row C_i and column E is a^i_{ef}, the number in the cell at row $\overline{C_i}$ and column $\overline{E}$ is a^i_{np}, the number in the cell at row C_i and column $\overline{E}$ is a^i_{ep}, and the number in the cell at row $\overline{C_i}$ and column E is a^i_{nf}).

Example 3.3.2. We give an example of a contingency table for the running example. Table 3.2 gives the data for the program spectrum of C_3.

As the total number of failing traces and the total number of passing traces are invariant for each $C_i \in \boldsymbol{PM}$, we have: $|\mathbf{F}| = a^i_{ef} + a^i_{nf}$, $|\mathbf{P}| = a^i_{ep} + a^i_{np}$, $|\mathbf{T}| = |\mathbf{F}| + |\mathbf{P}|$.

	E	$\overline{E}$
C_3	3	1
$\overline{C_3}$	0	6

Table 3.2: Contingency table

3.4 Probability Spaces

Definition 3.4.1. For each proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$, we identify a *probability space* with a tuple $\langle \mathbf{T}, \boldsymbol{K}, P_1 \rangle$, where

- $\mathbf{T}$ is the set of coverage vectors described as the *sample space*,
- $\boldsymbol{K}$ is the set of events described as the *event space*.
- P_1 is the *probability function* with signature P_1: $\boldsymbol{K} \rightarrow [0, 1]$.

$\boldsymbol{K}$ is defined inductively as follows: For each $C_i, C_j \in \boldsymbol{PM}$, $C_i, C_j \in \boldsymbol{K}$. If $X, Y \in \boldsymbol{K}$, then $X \cap Y$, $X \cup Y$, $\overline{X} \in \boldsymbol{K}$. P_1 is assumed to have the following standard properties of probability [117]. For all $X, Y \in \boldsymbol{K}$:

1. $P_1(X) = \frac{|X|}{|\mathbf{T}|}$
2. $P_1(X) = 1$ if $X = \mathbf{T}$.
3. $P_1(X) = 0$ if $X = \varnothing$.
4. $P_1(X \cup Y) = P_1(X) + P_1(Y) - P_1(X \cap Y)$.
5. $P_1(\overline{X}) = 1 - P(X)$.
6. $P_1(X|Y) = \frac{P_1(X \cap Y)}{P_1(Y)}$.

For clarity we shall drop the index of P_1 until chapter 6 (where we introduce a second probability function and will thereby need to disambiguate between them in our notation). Using the first property and the definition of a program spectrum, we observe that the following four identities hold: $P(C_i \cap E) = \frac{a^i_{ef}}{|\mathbf{T}|}$, $P(\overline{C_i} \cap E) = \frac{a^i_{nf}}{|\mathbf{T}|}$, $P(C_i \cap \overline{E}) = \frac{a^i_{ep}}{|\mathbf{T}|}$, $P(\overline{C_i} \cap \overline{E}) = \frac{a^i_{np}}{|\mathbf{T}|}$. Informally, the first identity states that the probability that the UTT corresponding to C_i is executed in a failing test case is the proportion of executions in which it is executed in a failing test case – conforming to a classical notion of probability. The other three identities are similarly explained.

Example 3.4.1. To illustrate a definition of a probability space, we define a probability space $\langle \mathbf{T}, \boldsymbol{K}, P_1 \rangle$ for the proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ in the running example. Recall, $\boldsymbol{PM} = \langle C_1, \ldots, C_4, E \rangle$ and $\mathbf{T} = \{t_1, \ldots, t_{10}\}$, where $C_1 = \{t_4, t_9, t_{10}\}$, $C_2 = \{t_1, t_5, t_{10}\}$, $C_3 = \{t_1, t_2, t_3, t_7\}$, $C_4 = \{t_2, t_6, t_7, t_9\}$, $E = \{t_1, t_2, t_3\}$. Using this setup we can find the probability of events in $\boldsymbol{K}$. For example, $P(E) = |E|/|\mathbf{T}|$ (by property 1), which is equal to $|\{t_1, t_2, t_3\}|/|\{t_1, \ldots t_{10}\}|$ (by the definition of E and $\mathbf{T}$). Thus, $P(E) = 3/10$. Informally, this states that the probability of the error is 0.3 – which represents the proportion of test cases in the test suite in which an error occurs, and intuitively describes the probability of a test case in the test suite failing.

Using the properties of probability and the above four identities it is possible to derive a measure for any probabilistic expression defined as a function of C_i and E. We provide an example of such a derivation below in Example 3.4.2. The results for many other derivations are presented in Table 3.2. The table can be used directly to transform probabilistic expressions defined as a function of C_i and E into a measure.

Example 3.4.2. We show how to derive $P(C_i) = \frac{a^i_{ef} + a^i_{ep}}{a^i_{ef} + a^i_{ep} + a^i_{nf} + a^i_{np}}$ using the above four identities and the above properties of probability. As follows $P(C_i) = P((C_i \cap E) \cup (C_i \cap \overline{E}))$ (by definition of set intersection). This is equal to $P(C_i \cap E) + P(C_i \cap \overline{E}) - P(C_i \cap E \cap C_j \cap \overline{E})$ (by property 4 of probability above). This is equal to $P(C_i \cap E) + P(C_i \cap \overline{E})$ (by property 3). This is equal to $\frac{a^i_{ef}}{|\mathbf{T}|} + \frac{a^i_{ep}}{|\mathbf{T}|}$ (by our identities). This is algebraically equal to $\frac{a^i_{ef} + a^i_{ep}}{|\mathbf{T}|}$, which is equal to $\frac{a^i_{ef} + a^i_{ep}}{a^i_{ef} + a^i_{ep} + a^i_{nf} + a^i_{np}}$ (by definition of $|\mathbf{T}|$).

Expression	Measure	Expression	Measure
$P(C_i)$	$\frac{a^i_{ef}+a^i_{ep}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(E\|C_i)$	$\frac{a^i_{ef}}{a^i_{ef}+a^i_{ep}}$
$P(\overline{C_i})$	$\frac{a^i_{nf}+a^i_{np}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(E\|\overline{C_i})$	$\frac{a^i_{nf}}{a^i_{nf}+a^i_{np}}$
$P(E)$	$\frac{a^i_{ef}+a^i_{nf}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(\overline{E}\|C_i)$	$\frac{a^i_{ep}}{a^i_{ef}+a^i_{ep}}$
$P(\overline{E})$	$\frac{a^i_{ep}+a^i_{np}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(\overline{E}\|\overline{C_i})$	$\frac{a^i_{np}}{a^i_{nf}+a^i_{np}}$
$P(C_i \cap E)$	$\frac{a^i_{ef}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(C_i\|E)$	$\frac{a^i_{ef}}{a^i_{ef}+a^i_{nf}}$
$P(\overline{C_i} \cap E)$	$\frac{a^i_{nf}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(C_i\|\overline{E})$	$\frac{a^i_{ep}}{a^i_{ep}+a^i_{np}}$
$P(C_i \cap \overline{E})$	$\frac{a^i_{ep}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(\overline{C_i}\|E)$	$\frac{a^i_{nf}}{a^i_{ef}+a^i_{nf}}$
$P(\overline{C_i} \cap \overline{E})$	$\frac{a^i_{np}}{a^i_{ef}+a^i_{ep}+a^i_{nf}+a^i_{np}}$	$P(\overline{C_i}\|\overline{E})$	$\frac{a^i_{np}}{a^i_{ep}+a^i_{np}}$

Figure 3.2: Basic probabilistic expressions and their measures

In addition to being able to derive a measure expressing the probability of any event defined as a function C_i and E, we can also use the above to find probabilistic expressions for many existing measures. See Example 3.4.3.

Example 3.4.3. We give an example to illustrate how a given measure, described in terms of the four elements of a program spectrum, can be equally described with probabilistic expressions. We use the Tarantula measure [133] as an example (described in the left hand side of the equation below). It is easy to show:

$$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}} = \frac{P(E|C)}{P(E|C)+P(E|\overline{C})}$$

This identification can be verified using Table 3.2.

3.5 Suspiciousness Measures

Definition 3.5.1. A *suspiciousness measure* w is a function with signature $w : \boldsymbol{PM} \to \mathbb{R}$, and maps each $C_i \in \boldsymbol{PM}$ to a real number as a function of C_i's program spectrum $\langle a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np} \rangle$, where this number is called the component's *degree of suspiciousness* [181].

The higher/lower the degree of suspiciousness the more/less suspicious C_i is assumed to be with respect to being a fault. Using our setup we can either express a suspiciousness measure in terms of the elements of a program spectrum directly, or alternatively in terms of a probabilistic expression which is itself expressible in terms of the elements of a program spectrum. Some additional notation is as follows, for all sets of components $X, Y \subseteq \boldsymbol{PM}$ we write $X >^{\mathbf{T}}_{w} Y$ just in case every element in X is more suspicious than every element of Y using suspiciousness measure w and set of coverage vectors $\mathbf{T}$. It will often be convenient to write $w(a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np})$ as shorthand for $w(C_i)$. Tables of established suspiciousness measures are given in Appendix A. Discussion of many measures is given in the literature review of chapter 2. An example is given in section

3.6 Properties

In this section we present and discuss some established properties of suspiciousness measures. A first general assumption which is established in the literature is as follows [213]: The prior probability distribution of faultiness is unknown, and therefore not exploitable by a suspiciousness measure. Despite the distribution of faultiness not being known, some suspiciousness measures satisfy specific formal properties, which are argued to be advantageous to fault localisation or certain fault localisation sub-problems. In the remainder of this section we describe these properties.

First, a prominent property is *rationality* [179, 181]. A suspiciousness measure w is *rational* if and only if the following two properties are satisfied. Where $c \in \mathbb{N}$ and $c > 0$

1. $w(a_{ef}, a_{nf}, a_{ep}, a_{np}) \leq w(a_{ef} + c, a_{nf} - c, a_{ep}, a_{np})$
2. $w(a_{ef}, a_{nf}, a_{ep} + c, a_{np} - c) \leq w(a_{ef}, a_{nf}, a_{ep}, a_{np})$

A stronger property than rationality is the property of *strict rationality*, which is obtained by replacing $\leq$ with $<$ in the above definition. Roughly speaking, a measure is rational/strictly rational if more failing traces covering a component lead to it being marked as more suspicious, and more passing traces covering a component lead to it being marked as less suspicious, conforming to our intuition of suspiciousness. Many suspiciousness measures have been shown to have strict rationality, at least when $a_{ef}, a_{ep} > 1$ [179, 181]. Naish et al argue that it is reasonable to restrict the SBFL approach to rational measures [182]. We assume this too in this thesis. An example of strictly rational and rational measures is given in Example 3.6.1.

Example 3.6.1. The Wong-II measure $a_{ef} - a_{ep}$ is strictly rational. Many measures, such as the Tarantula measure $\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$, are rational but not strictly rational due to their behaviour when $a_{ep} = 0$. To see this, we observe that the Tarantula measure returns the same value of 1 whenever $a_{ef} > 0$ and $a_{ep} = 0$, thus violating the first condition of strict rationality. See [179, 181] for discussion.

A second property is *single-fault optimality* [179, 181], which is based on the observation that if a program contains only a single fault then all failing test cases must cover that fault. In general, for a given proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ we say a technique g is *single fault optimal* if for all $C_i, C_j \in \boldsymbol{PM}$, if $a^i_{ef} = |\mathbf{F}|$ and $a^j_{ef} < |\mathbf{F}|$ (or $a^i_{ef} = a^j_{ef} = |\mathbf{F}|$ and $a^i_{ep} < a^j_{ep}$) then C_i is more suspicious than C_j using g. Given it is assumed that UUTs are investigated by the user in descending order of suspiciousness of their corresponding components, we then have the following definition of single-fault optimal SBFL measures. According to Naish et al [179, 181], a suspiciousness measure w is *single-fault optimal* if it satisfies the following two conditions.

1. If $a_{ef} < |\mathbf{F}|$, then the value returned is less than any value returned when $a_{ef} = |\mathbf{F}|$, and
2. If $a_{ef} = |\mathbf{F}|$ and $a_{np} = k$, then the value returned is greater than any value returned when $a_{np} < k$.

A framework that optimises any given SBFL measure to being single fault optimal was first given by Naish in [182]. For a suspiciousness measure w scaled from 0 to 1 and input vector $\vec{x} = \langle a_{ef}, a_{nf}, a_{ep}, a_{np} \rangle$, we can construct the *single fault optimised* version for w (written $Opt(w)$) as follows (here, we use the equivalent formulation of Landsberg et al [149]):

$$Opt(w) = h, \text{ where } \begin{cases} h = a^i_{np} + 2 & \text{if } a^i_{ef} = |\mathbf{F}| \\ w(C_i) & \text{otherwise} \end{cases}$$

Single bug optimisation can be extended to any fault localisation method m which returns the index of a most suspicious program component in $\boldsymbol{PM}$. We define $Opt(m)$ as follows. Let $S = \{C_i \in \boldsymbol{PM} | a^i_{ef} = |\mathbf{F}|\}$. If $S \neq \varnothing$, return the index of a component in S covered by the fewest passing traces, otherwise return the index returned by m.

Finally, we discuss the property of ranking equivalence. We begin with the definition for when measures are monotonically equivalent. Here the notation $\vec{x}$ symbolises a program spectrum.

Definition 3.6.1. Two suspiciousness measures w_1 and w_2 are *monotonically equivalent* if and only if $(\forall \vec{x}, \vec{y})\ w_1(\vec{x}) < w_1(\vec{y}) \Leftrightarrow w_2(\vec{x}) < w_2(\vec{y})$

Naish et al. proved that many groups of suspiciousness measures are monotonically equivalent on domains in which the measures share the same proband model [181]. In other words, they proved they were monotonically equivalent in proband models in which the number of failing test cases $\mathbf{F}$ and the number of passing test cases $\mathbf{P}$ are the same for the spectra $\vec{x}$ and $\vec{y}$. This property is called *ranking equivalence* [181]. Intuitively, ranking equivalent measures are equivalent for ranking components in terms of suspiciousness. Thus, two measures are ranking equivalent just in case, for any proband model, they rank the same components as most suspicious, the same components as second most suspicious, the same components as third most suspicious, etc. Ranking equivalence proofs have been given for many SBFL measures [58,149,181]. An example of a pair of ranking equivalent measures is given in Example 3.6.2.

Example 3.6.2. The Tarantula measure $\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$ is ranking equivalent to the simpler measure $\frac{a_{ef}}{a_{ep}}$ (see [181]). Naish et al identify 6 classes of ranking equivalent measures $S1 - S6$. These classes are grouped in Tables A.1 and A.2 in Appendix A.1.

3.7 SBFL method

In this section we describe the established SBFL algorithm. The method produces a list of program component indices ordered by suspiciousness, as a function of set of coverage vectors $\mathbf{T}$ (taken from a proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$) and suspiciousness measure w. Versions of this algorithm are presented in [13, 133]. As the algorithm

C_i	a^i_{ef}	a^i_{nf}	a^i_{ep}	a^i_{np}	$w(C_i)$
C_1	0	3	3	4	0.00
C_2	1	2	2	5	0.54
C_3	3	0	1	6	0.88
C_4	1	2	3	4	0.44

Table 3.3: Suspiciousness scores for running example using Tarantula

is simple, we informally describe the algorithm in three stages, as follows. First, the program spectra for each program component is constructed as a function of $\mathbf{T}$. Second, the indices of program components are ordered in a *suspiciousness list* according to decreasing order of suspiciousness. Third, the suspiciousness list is returned to the user. In a semi-automated single fault localisation paradigm, the user will inspect each UUT corresponding to each index in the suspiciousness list in decreasing order of suspiciousness until a fault is found. In a semi-automated multiple fault localisation paradigm, the user will inspect each UUT corresponding to each index in the suspiciousness suspiciousness list in decreasing order of suspiciousness until all (or a given number of faults) are found.

Example 3.7.1. We illustrate an instance of SBFL using our running `minmax.c` example of 3.1, and the Tarantula measure as an example suspiciousness measure. First, the program spectra (represented by the second column of Table 3.3) is constructed as a function of the given coverage vectors (represented by the coverage matrix of Table 3.1). Second, the suspiciousness of each program component is computed (last column of Table 3.3), and ordered according to decreasing order of suspiciousness. Using Table 3.3 we get the list $\langle 3, 2, 4, 1 \rangle$. Finally, the list is returned to the user, and the lines of in the program are inspected according to this list in descending order of suspiciousness until a fault is found. In our running example, `C3` is investigated first, and thus the fault is found immediately by the user.

Chapter 4

Experimental setup

In this chapter we present the experimental setup common to the experiments performed in chapters 5, 6, and 7. In general, we perform two different types of experiments, which are informally stated as follows:

1. *Single fault localisation task* (SFL). Here the task is to experimentally compare the effectiveness of different techniques at the task of finding a single fault in a program (Chapters 5 and 6).

2. *Multiple fault localisation task* (MFL). Here the task is to experimentally compare the effectiveness of different techniques at the task of finding multiple faults in a program. (Chapter 7).

In the remainder of this section we describe benchmarks and evaluation methods.

4.1 Benchmarks

We use two sets of benchmarks, which we call the SIR and Steimann (STI) benchmarks respectively. The former consist of small programs, the latter of large programs. We designed our experiments to ensure it is (to our knowledge) one of the largest scale experiments in software localisation to date.

4.1.1 Small programs

Benchmark	LOC	b	LOC/b	t	1v	2v	3v	4v
`print_tokens`	478	49	10	4130	6	14	16	9
`print_tokens2`	399	55	7	4115	7	19	21	16
`replace`	512	79	6	5542	22	203	50	36
`schedule`	292	38	7	2650	8	34	33	55
`schedule2`	301	32	9	2710	9	30	76	85
`tcas`	141	13	10	1608	28	89	71	63
`tot_info`	440	32	14	1051	19	167	94	91
`gzip`	7996	32	250	214	4	6	4	1
`sed`	11990	104	115	360	5	9	7	2
`space`	6218	254	24	13585	23	68	46	20

Table 4.1: SIR benchmarks

In this section we describe the SIR benchmarks used in our experiments. The SIR benchmarks are divided into the Siemens test suite (which consists of `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`), and three larger programs `gzip`, `sed`, `space`. All these are are C programs. Table 4.1 describes these benchmarks. The first column gives the name of the benchmark, the second the number of lines of code (LOC) in the "golden" (correctly working) version of that benchmark, b is the average number of blocks of code (where a block corresponds to a maximal set of lines of code with the same test cases covering them), LOC/b the average number of lines of code per block, and t the number of test cases that were supplied with the benchmark. 1v, 2v, 3v, 4v give the number of single, double, triple, and quadruple fault versions generated for the benchmark. Each benchmark has a single correct working "golden" version of the program, a number of faulty versions of the same program, and a number of test cases in a test suite (usually designed to maximise coverage of the program). In the original benchmarks available from the SIR repository [7], the faulty versions usually contained only a single fault. Details of the benchmarks are given below.

The Siemens programs were assembled at Siemens Corporate Research in order to study how software fault localisation could take place as a function of coverage data extracted from program executions (see [120]). The remaining programs (the bottom three of Table 4.1) were collated by SIR. The programs in the suite perform a variety of different functions:

1. `tcas` is an aircraft collision avoidance system.

2. `schedule2` and `schedule` are priority schedulers.

3. `totinfo` computes types of statistics for certain data.

4. `printtokens` and `printtokens2` are lexical analysers.

5. `replace` performs pattern matching and substitution.

6. `space` is an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, `space` outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages. The original test suite for `space` was constructed by Vokolos et al. [227].

7. `gzip` and `sed` are both UNIX utilities. `gzip` is a file format and a software application used for file compression and decompression, and `sed` is a stream editor which parses and transforms text.

We now discuss how faulty versions were created for the original benchmarks by the Siemens team and contributors to SIR [7]. Faulty versions of the Siemens programs were created by manually seeding those programs with faults, usually by modifying a single line of code in the program (although sometimes by inserting or deleting a line of code) in the golden version. The aim was to introduce faults that were as realistic as possible, based on their experience with real programs. Test suites were created for each program. To make these test suites, an initial suite of black-box test cases were created according to good testing practices, based on the tester's understanding of the program's functionality, using the category partition method and the Siemens Test Specification Language tool. They then augmented this suite with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test cases. To obtain meaningful results with the seeded versions of the programs, the researchers retained only faults that were neither too easy nor too hard to detect, which they defined as being detectable by at most 350 and at least 3 test cases in the test pool associated with each program.

As the original versions of the benchmarks (made available at the Software-artifact Infrastructure Repository (SIR) [7]) usually only contained a single bug, Abreu extended these benchmarks to the multiple fault case [13, 21]. Accordingly,

each "golden" version of a program was extended with the faulty versions in such a way as arbitrary combinations of faults could be activated using macro definitions. Thus, any combination of the original faults associated with the original single fault versions could be activated simultaneously. The set of original faults made available were limited to a selection of 143 out of the 183 original faults, based on criteria such as faults being attributable to a single line of code, in order to enable unambiguous evaluation. We used these versions of the SIR benchmarks in our experiments.

We now describe how we generated faulty versions using Abreu's version of the SIR benchmarks. For each benchmark we randomly generated 100 (or an exhaustive number) of 1 fault versions, 100 (or an exhaustive number) of 2 fault versions, and so on for the 3 and 4 fault versions. We then removed versions if there was a fault that was not covered by a failing test case. To illustrate, using this process we obtained for the `tcas` program – 28 faulty versions which covered 1 fault, 89 versions which covered 2 faults, 71 which covered 3, etc., as detailed by Table 4.1.

For each faulty version, there was a unique test suite associated with that faulty version (which took the form of a text file consisting of input vectors). The exception was for `space`, which had many different test suites associated with it. We used `testsuite1`, which is a set of 150 input vectors associated with the space program [21]. It was not feasible to run all 10k+ test cases supplied on `space`. Each faulty version and test suite constituted a proband. We now discuss error detection. A test case counted as failed if its output differed from the corresponding output of the supplied "golden" program version, and as passed otherwise (In this way, the correct version supplied us with a virtual specification for the faulty version). The faulty lines of code were identified in the source-code for us.

We now discuss how we created a proband model for each proband. For each test suite and associated faulty program version, we used the Gcov tool to extract coverage information for each test case executed on the faulty program. Gcov is a tool used in conjunction with the GCC compiler designed to inform the user what lines of code are actually executed in a given test case, and can additionally provide a variety of profiling information [3]. After all the coverage information was gathered for a given proband, we generated the associated set of coverage vectors, which was then stored in a text file in a form that represented a coverage matrix. The granularity of our UUT was chosen to be *coverage blocks* (from hereonin *blocks*), where a block is defined as a maximal set of executable statements with the same test cases covering them. Each coverage matrix was stored in such as way as to encode the details of our proband model – with rows of the matrix representing each coverage vector in the

set, and each column associated with a program component in the program model (as per the setup of chapter 3). For all our experiments, a block was considered to be *faulty* just in case one of the lines of code it covered was faulty, and thus we could identify faulty components.

We now discuss the choice of coverage blocks as the level of granularity for our experiments. In practice, coverage blocks provide a natural grouping for lines of code, as the degree of suspiciousness of lines of code will always be the same as the block to which they belong, and so does not affect the fault localisation process from the user's point of view (consequently a measure will received better W-scores using coverage blocks just in case it receives better W-scores using statements as the level of granularity). It also saved a great deal of time as far as running the experiments was concerned (compare the number of blocks with the number of lines of code in either of the tables of benchmarks above), making the scale of the experiment feasible to perform. In the majority of our benchmarks, coverage blocks represented a continuous chunk of the program and were roughly similar in size – the average size of these blocks are reported in the benchmark Tables 4.1 and 4.2.

4.1.2 Large programs

Benchmark	M	UUT	b	UUT/b	t	1/2/4/8/16/32v
`Daikon 4.6.4`	14387	1936	48	40	157	353/1000/1000/...
`Eventbus 1.4`	859	338	68	5	91	577/1000/1000/...
`Jaxen 1.1.5`	1689	961	228	4	695	600/1000/1000/...
`Jester 1.37b`	378	152	25	6	64	411/1000/1000/...
`Jexel 1.0.0b13`	242	150	48	3	335	537/1000/1000/...
`JParsec 2.0`	1011	893	240	4	510	598/1000/1000/...
`AC Codec 1.3`	265	229	57	4	188	543/1000/1000/...
`AC Lang 3.0`	5373	2075	78	27	1666	599/1000/1000/...
`Eclipse.Draw2d 3.4.2`	3231	878	74	12	89	570/1000/1000/...
`HTML Parser 1.6`	1925	785	148	5	600	599/1000/1000/...

Table 4.2: Steimann's benchmarks

Our second set of benchmarks is provided for by Steimann et al. (henceforth "Steimann" suite) [213], described in Table 4.2. All the programs described in this suite are Java programs. The first column gives the name of the benchmark, the second the number of methods (M) for the golden version of that benchmark, UUT gives the number of units under test, b gives the average number of coverage blocks for a program version, UUT/b gives the average number of units under test per block, and

t the number of test cases. 1/2/4 ... gives the number of single, double, quadruple, ... etc. fault versions respectively. Given that each method contains many lines of code, the benchmarks here are much larger than the ones in the SIR test suite (e.g. Daikon is known to have 100k+ LOC). Test suites for the benchmarks were created using JUnit. Here, the length of the benchmarks were described by Steimann et al. in terms of methods, as opposed to lines of code [213]. To obtain the number of methods they used the Eclipse Java search facility, which "provided more accurate counts than any metrics tool that we tried (whose counts did in fact vary widely, presenting a threat in its own right" to the validity of the experiment.

We now discuss how we generated our proband models. Steimann et al provided us with a C program and data repository that automatically generated coverage matrices (along with the identified faults) as a function of the repository. The data repository stored coverage information about the test cases for a given program version, and the step of compiling the original 50k+ Java programs and extracting the coverage data itself was performed by Steimann et al, and is a step which was infeasible to complete ourselves for the purposes of this thesis given the amount of time this would take. Finally, we adapted their C program to create coverage matrices at the coverage block level of granularity, and stored in text files in the same format as discussed in the previous subsection.

Some general descriptions of the golden versions of the benchmarks are as follows:

1. `Daikon` is designed to detect likely invariants of programs, where an invariant is a condition that always holds true at certain points in the program. It is mainly used for debugging programs in late development, or checking modifications to existing code.

2. `Eventbus` is a publish-subscribe application programming interface (API) in which subscriptions are based on class semantics and/or string matching. It is used to simplify communication between threads in Java programs.

3. `Jaxen` is described as a universal object model walker, capable of evaluating Xpath expressions, where Xpath is a syntax for defining parts of an XML document.

4. `Jester` is a lightweight operating system.

5. `Jexel` provides integration of Microsoft Excel into Java applications.

6. `JParsec` is a parsing program.

Name	Function
Negate Decision (ND)	negate condition in an if or while statement
Replace Constant (RC)	replace integer constant C by 0, 1, -1,C+1 or C-1
Delete Statement (DS)	delete a statement
Replace Operator (RO)	replace an arithmetic, relational, logical, bitwise logical, increment/ decrement or arithmetic-assignment operator by an operator from the same class
Assign Null (AN)	replace rhs of assignment with null
Return Null (RN)	replace return expression with null

Table 4.3: Fault injection methods

actual faults	1	2	4	8	16	32
covered faults	1.00	1.92	3.63	6.71	11.81	20.02

Table 4.4: Fault injection methods

7. `AC Codec` (Apache Commons Codec) software provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.

8. `AC Lang` (Apache Commons Lang) provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and system properties.

9. `Eclipse Draw` is a lightweight tool-kit for displaying graphical components on an SWT Canvas, where the Standard Widget Tool-kit (SWT) is a graphical widget tool-kit for use with the Java platform.

10. `HTML Parser` is a program for automated Hypertext Mark-up Language (HTML) parsing.

We refer the reader to [213] for in-depth discussion and details of the benchmarks. We now discuss how faulty versions were generated by the original authors. Each version of a benchmark with n faults was generated automatically using fault injection methods. The set of mutations are presented in Table 4.3 and discussed in [213]. We observed that not all faults were covered by test cases supplied by a proband. In Table 4.4 the average number of faulty components that were covered for all 1/2/4/8/16/32 fault benchmarks was found are given in the succeeding row. On average, a program version contained 7.52 covered faults.

We now discuss the major differences between the SIR and Steimann benchmarks. The major difference is their scale, and quality. We discuss these differences as follows. First, scale. The Steimann suite is much larger in three dimensions:

1. The programs were larger (100+ methods, whereas programs in SIR were usually under 1000 LOC)

2. The sample sizes of the n-faulty versions of each benchmark was larger (usually 1000 versions, whereas with SIR there were usually less than 100)

3. The suite could represent a larger range of faults per program (1–32, whereas SIR was 1-4).

Accordingly, we refer to the Steimann benchmarks as our "large" benchmarks, and to the SIR benchmarks as our "small" benchmarks.

Conditions (1) and (2) were out of our control, as these were fundamental properties of the benchmarks as given. The reason for (3) was that for SIR benchmarks we had to limit ourselves to four faults. This was because we wanted to generate as many n fault benchmarks as could be uniformly generated across the suite, and as the `gzip` benchmark only had four single fault versions, it only became possible to generate a maximum of four faults for that benchmark, and hence the others. Versions from this test suite in which faults were not covered by a failing test case, were not included. In contrast, for the Steimann benchmarks not all the faults were covered by a failing test case in many of the versions. We did not delete program versions that did not cover all faults in the larger benchmarks because this was so frequent, especially for those with a high number of faults.

We now discuss differences in quality between the large and small benchmarks. The quality of injected faults is very different. The SIR benchmarks contain faults which are manually injected (and thus more natural looking), whereas the Steimann benchmarks are automatically injected. We can find no reason for thinking that this means that the SIR benchmarks are better quality for the purposes of a fault localisation experiment. This is because all our fault localisation methods are a function of the coverage information of test suites, and because the particularities about given faults are hugely abstracted over, their original nature becomes more irrelevant. Furthermore, the automatically injected faults, as described by Table 4.3, do some work in simulating some types of common mistakes made by engineers, or at least simulate many of their effects.

Another difference involved the quality of the coverage matrices. For the SIR benchmarks, test cases were designed in terms of maximising coverage and were a function of input vectors to the entire program itself. In contrast, the test cases for the Steimann benchmarks were made using JUnit [213], can be used execute and test individual functions within the entire program. This had the potential to make failing test cases which covered relatively few program components. Another important difference is the fact the small benchmarks consist of C programs, and the large consist of Java.

4.2 Evaluation Methods

In this section we present our methods for evaluating how well a given measure performs as a fault localiser. We limit ourselves to discussion of the task of locating a *single* fault in a program (in a program with any number of faults) here, as our evaluation criteria for our multiple fault localisation experiments will require presentation of the particular multiple fault localisation algorithms themselves for them to be clearly understood. We use two major means of assessment: a wasted effort score (W-score), and an absolute ranking score (A-score). We also perform Wilcoxon rank-sum significance tests using these scores. In addition, we discuss how to estimate the scores for an ideally performing strictly rational SBFL measure (scores which no strictly rational SBFL measure can exceed).

4.2.1 Wasted effort scores

In the context of a practical semi-automated fault localisation task on a given faulty program, a fault localisation tool is assumed to be effective if, after the tool's fault localisation information has been passed to the user, the amount of effort an engineer wastes looking for faults is otherwise reduced. Furthermore, the degree of effectiveness of a tool may be measured in terms of the amount of wasted effort. One established way to estimate wasted effort is to calculate the percentage of non-faulty blocks of code an engineer would be expected to look through until a fault is found using the given tool [18, 181, 241]. This percentage is called the wasted effort score (W-score), and an effective method is expected to have low wasted effort scores on average. In this section we present the definitions for W-scores.

We begin by stating that underlying our W-scores are two conventional assumptions established for the purposes of evaluation [213]:

1. We assume the user will always recognize a component considered to be faulty in the context of the evaluation to be faulty.

2. We assume the user will never recognize a component considered to be non-faulty in the context of the evaluation as faulty.

In developing a formal definition of a W-score, we consider the subtlety of how to score in cases where a most suspicious faulty block is tied with other faulty or non-faulty blocks. One approach is to score the technique in the best case scenario in which the bug is investigated first in the case of ties, which leads to a *best* case score. This can be informally described as the percentage of non-faulty blocks with strictly *greater* suspiciousness than the most suspicious bug. Formally, where w is the given suspiciousness measure we wish to provide a wasted effort score for, and B is known as faulty in the program model $\boldsymbol{PM}$ with the highest degree of suspiciousness of any bug according to w, then

$$best(w) = \frac{|\{C_i \in \boldsymbol{PM} | w(C_i) > w(B)\}|}{|\boldsymbol{PM}|} 100$$

Here we exclude E from the program model $\boldsymbol{PM}$ (so that only components corresponding to UTTs are considered). An alternative approach to best scores is to score the measure in the worst case scenario in which the bug is investigated last in the case of ties, which leads to a worst case score. This may informally be described as the percentage of non-faulty program components with a *greater or equal* suspiciousness than a most suspicious bug. Formally,

$$worst(w) = \frac{|\{C_i \in \boldsymbol{PM} | w(C_i) \geq w(B)\}| - 1}{|\boldsymbol{PM}|} 100$$

Both *best* and *worst* scoring methods as defined have been consistently used in the literature [18, 181, 241]. We argue that both these scoring methods have issues. We argue the *best* score is problematic, since the optimal measure under this scoring method is the measure which assigns the same degree of suspiciousness to each component. However, this measure is obviously useless to the user in practice as it does nothing to discriminate more from less suspicious components, and so should not be regarded as scoring well.

In contrast, we argue that the *worst* score is also problematic, since a measure would be expected to increase the mean of its scores over many program versions simply by adding a small random number to otherwise equally suspiciousness scores, thereby breaking any ties, and improving the measure's score over a sufficiently large sample. This positively biases measures which break ties at random.

An alternative to worst and best scoring methods which allays much of these problems is to take the mean. For instance, the following mean scoring method (here expressed as a percentage) is proposed by Abreu [13]:

$$mean(w) = \frac{best(w) + worst(w)}{2}$$

The assumption behind this is that if there is only *one* fault in the program, then we'd expect half of the non-faulty blocks which are equally suspicious to the most suspicious fault to be inspected before a faulty block is found. However, this too is problematic if we use the scoring method with programs with multiple bugs. When there are *many* bugs otherwise equally ranked with the most suspicious bug, measures which break ties at random would be expected to be favoured in terms of their mean scores over a large sample, therefore biasing measures which break ties at random. Consequently, we argue that a better scoring method than the best, worst, and mean scoring methods is one which measures the *expected* percentage of non-faulty blocks the user would have to inspect before finding a bug as a function of the number of bugs in the program. Formally, where *bugs* is the number of bugs with the same suspiciousness score as the most suspicious bug b, then

$$average(w) = best(w) + \frac{worst(w) - best(w)}{bugs + 1}$$

A consequence of this scoring method (which the other measures do not satisfy) is that in a program with 2, 3, 4 ... etc bugs, the expected percentage of non-faulty blocks a user would have to investigate using a measure which ties all components would be just under 50%, 33%, 25% ... etc respectively, which is just what we would expect in practice. Our average measure thus provides a better analysis of the expected percentage of non-faulty blocks one would have to inspect before finding a fault using a given method. We use this definition of a W-score in our experiments.

4.2.2 Absolute ranking scores

We performed a second means of assessment using *absolute scores* (A scores). Following Parnin and Orso [188], absolute scores are designed to gauge how effective a method is in terms of whether it ranks a fault in the top "handful" of most suspicious UUTs. Accordingly, for a given program version a measure receives 100% if a faulty block was a member of the set of the n most suspicious blocks (and 0% otherwise), for some cut-off of n. In our experiments we use $n = 1$, which meant that our A scores measured whether a technique identified the most suspicious component (and therefore the first component investigated by the user) with a fault. We think measuring whether a method behaves perfectly (i.e. identifies the most suspicious component as a fault) is a useful indicator of effectiveness. An additional reason for our low cut off for n was because coverage blocks can be quite large (already containing a number of methods or lines of code) and faults could often be abundant (especially in our large benchmarks), and therefore it can often be reasonable to expect a good fault localisation method to locate a fault in the most suspicious block in the context of our experiments.

We first discuss the difference in importance between W and A scores as evaluation measures. In recent years there has been the question as to which is the better way to assess the effectiveness of a SBFL measure, with many authors now preferring to supply both types of scores in some form [149, 165]. Parnin and Orso [188] conclude that A scores are more useful for assessing SBFL methods at the task of finding a single fault. They base this conclusion on the fact that in their study engineers were prone to ignore suspiciousness lists after investigating the top handful of suspicious components. This conclusion is based on observations from a study they performed of 20 programmers who used the Tarantula tool on two small (2-4k LOC) single bug programs [188]. We argue the conclusion of Parnin and Orso is controversial – just because engineers in their study began to ignore suspiciousness lists after a few investigations, this does not automatically entail that they *should*. Instead, it might equally be argued that a conclusion of that study is simply that the engineers simply did not use the particular tool correctly (as prescribed).

Thus, we address the question: what is more important in evaluating a technique's effectiveness – W or A-scores? We think the answer to this question comes from considering what the technique is designed to do in the first place. For instance, in contexts in which only a single top suggestion can be used (e.g. a measure is being used as a subroutine in an automated tool such as Genprog [98], to find where a program

repair algorithm should uniquely focus its repair), then A-scores are preferred as the measure of effectiveness. In contrast, if a user is investigating a large amount of faulty but ill-understood code in the semi-automated fault localisation paradigm, the user might be advised to investigate it in order of suspiciousness until a fault is found, in which case W-scores would be preferred. In conclusion, we suggest both scoring methods are good for evaluating a given measure for different tasks. As we are more interested in the semi-automated fault localisation paradigm, we think W-scores are more important for evaluating techniques in terms of effectiveness for the purposes of this thesis.

We raise one issue about A-scores which we believe has a bearing on how a measure is valued. To see the issue, consider a measure w_1 which always isolates a fault in the $n + 1th$ investigated component, where we are using an A-score with a cut-off of n. This would automatically be the worst possible measure according to A-scores with a cut off of n (receiving a mean A-score of 0). Accordingly, a measure w_2 which ranked all bugs as the last component to be inspected, except for one occasion in which it isolated a fault in the n-th, would be superior to w_1 (because its mean A-score would be greater than 0). A-scores therefore lose the ability to score a measure in terms of its proximity to perfect performance, a factor which W-scores accommodate well.

Finally, we observe that our W scores and A scores have the following relationship - a measure w received an W score of 0% for that program version if and only if it received 100% in terms of A score. Intuitively, this is because if no effort has been wasted (i.e. no-non faulty blocks have been investigated) in finding a fault, then the first block investigated was faulty, and vice versa.

4.2.3 Upper and lower bound scores

In this section we discuss how to estimate an *upper bound* for the scores for the SBFL approach. Intuitively, what we call upper bound scores are scores which no SBFL measure can exceed. Naish et al. present upper bound scores in the form of what is defined as the *unavoidable costs* for any strictly rational measure. These are the scores that the best performing strictly rational measure can possibly receive [182]. To determine this score, one constructs an ordered list with the property that for every component, C_i is ranked higher than a given fault C_j just in case every strictly rational measure would rank C_i higher than C_j. The W/A-scores of this list are the unavoidable cost W/A-scores. Following Naish et al, we take unavoidable cost scores to reasonably estimate the upper bound limit for the performance of the SBFL

approach in general (see [182] for details). Upper bound scores are particularly useful in determining whether competing techniques can outperform the best possible scores of SBFL, and will be of interest in Chapter 6 when we compare SBFL with our new rival framework.

We now present a way to estimate the *lower bound* scores for a technique. In theory, lower bound scores are scores which no reasonable SBFL measure should score lower than. To establish a baseline for localisation efficiency, we included a random measure (called "Random") which assigns each program component a random suspiciousness score. Informally, the random measure models a completely uninformed engineer who inspects the program at random hoping to find a fault. Consequently, we would reasonably expect suspiciousness measures to perform better than this measure. To get the random score, we ran each experiment 10 times with the random measure and took the average.

4.2.4 Higher level scores

We define general performance indicators in the form of *higher level* W (or A) scores. For each suite (SIR/Steimann), there are ten benchmarks, where:

1. The score for the *n-fault version of a benchmark*, is the mean of the scores of the n-fault versions of that benchmark. For example, the score for the 2-fault versions of the Daikon benchmark, is the mean of the 1000 scores received for that benchmark. Each score for an n-fault version of a benchmark is simply called a *score*. Each technique has 100 scores in total, 60 for the large benchmarks and 40 for the small.

2. The score for the *n-fault versions of a suite*, is the mean of the scores of a suite. E.g. the score for the 2 fault versions for the Steimann suite, is the mean of the 2 fault version of Daikon, the 2 fault versions of Eventbus, etc.

3. To generate overall *AVG scores for a suite*, we take the mean of the scores in that suite. For example, the AVG score for Steimann benchmarks is the mean of its 60 scores.

4. To generate overall *AVG scores*, we take the mean of all the scores of each suite (there are 100; 60 from the Steimann benchmarks, 40 from the SIR benchmarks).

A consequence of the above scoring method is that each score is weighted equally in the final AVG, as is established convention [181]. A good technique will have low AVG W-scores and high AVG A-scores. We emphasise that higher-level scores are no more than coarse grained indicators. We also emphasise that, to our knowledge, no better ways to assess the performance of ranking based techniques on a set of benchmarks has yet been presented in the literature.

4.2.5 Significance tests

We performed a final means of assessment using *Wilcoxon rank-sum tests*. The Wilcoxon rank-sum test is a non-parametric statistical test which tests whether one population of values is significantly larger than another population [234]. A non-parametric test was chosen because the underlying distribution is assumed to be unknown. Using this test, we were able to establish which measures' W or A-scores were significantly better than others.[1]

In addition, in order to assess the relationship between the number of bugs in a program and a given measure's W-score, we used the Spearman's rho measure [210]. Spearman's rho measures how well the relationship between two variables can be described using a monotonic function. Here, the variables in question were the number of bugs (1, 2, 3, 4 for the SIR benchmarks, 1, 2, 4, 8, 16, 32 for the Steimann benchmarks), and the W-scores the given measure received on average for the program with the given number of faults. A significant relationship was obtained with an R-score of -0.99 or less.

4.2.6 Evaluation

Using the evaluation methods described in this section, we can now state how we evaluate the claim that a given method is effective and efficient at single fault localisation. Following the literature review of chapter 2, established methods at the state of the art of statistical fault localisation have been demonstrated to be effective at finding faults, where effectiveness has typically been measured in terms of that method's W or A scores obtained in large scale experimentation. Accordingly, in this thesis we empirically substantiate the claim that a new technique is effective at a given fault localisation task by showing it improves on the AVG W and A scores of

[1]Following a convention on small sample sizes, we applied continuity correction by adjusting the Wilcoxon rank statistic by 0.5 towards the mean value when computing the z-statistic.

state of the art techniques. Furthermore, we empirically substantiate the claim that a technique (for localizing either single or multiple faults) is efficient by demonstrating it maintains a low average runtime in large scale experimentation, and assume that a runtime of ten seconds or less on average is sufficient to count as low (given this is negligible amount of time for the user to wait).

Discussion of how to evaluate the claim that a given method is effective at multiple fault localisation is postponed to section 7.6, as this will require some more detailed discussion of multiple fault localisation methods.

4.3 Threats to Validity

In this section we discuss threats to the validity and value of any results obtained the experimental setup described in this chapter. Threats to the validity of empirical studies of SBFL techniques is a research topic in its own right [213]. In this section we summarise some of these threats. The underlying worry for most of them is the inability of the experimental results to generalise more broadly as predictors of the techniques' accuracy or usefulness in practice. Issues include the following:

1. The faults may be unrepresentative. The programs in our large benchmarks have automatically injected faults, which may mean they are not always representative of natural faults in the real world. In addition, certain types of faults (such as faults of omission) were not included and the distributions of how many faults there are in programs may vary from context to context.

2. The programs may be unrepresentative. Some of the programs may not be generally representative of real-world programs, given the variety of languages, sizes and code quality which exists.

3. The test suites may be unrepresentative. In real-world testing scenarios the test suites vary in terms of size and quality and in our benchmarks the test suites had good coverage which advantages the process of fault localisation.

4. The data gathering stage may be unrepresentative. The conventional method of gathering coverage data relies upon comparing executions between golden and faulty mutants of a program, which is often unlike real-world scenarios.

5. The evaluation methods may be unrepresentative. We make the conventional assumption that a programmer always recognizes the status of a piece of code

as faulty or non-faulty when investigating the program. In practice, this may not be the case. A related criticism is that our evaluation methods do not necessarily measure how quickly a programmer will find faults in practice.

6. Coverage blocks are large. The average size of the coverage blocks are reported in the benchmark Tables 4.1 and 4.2, and can often be quite large, suggesting that even if a fault is ranked highly by a measure the user will still have to look through a large amount of code, suggesting that fault localization techniques could be improved wrt accuracy.

We emphasise that the above threats are faced by many SBFL studies in general [213]. As a response, we agree with Steimann et al. who argue that experiments such as ours play a necessary role in evaluating the accuracy of a technique and that practical utility depends on a technique's accuracy. Indeed, as Steimann et al. conclude – "who would deny the value of a fault locator with perfect accuracy?" [213]. Moreover, a superior method to experimentally assess SBFL techniques which is at once sufficiently scaled and feasible has not yet been found.

Given the existence of threats to the validity and value of empirical assessments of fault localisation methods, we agree with Steimann et al that the theory which motivates a given technique to the fault localisation problem becomes relatively more important [213]. In conclusion, we believe that the best techniques will have a good balance of empirical performance (low W-scores and high A-scores) and satisfy desirable formal properties motivated as key to fault localisation.

Chapter 5

Spectrum-based Fault Localisation

In Chapter 1 we identified SBFL as one of the most prominent, efficient, and effective current methods of software fault localisation. One of the major projects of SBFL research is to find new and better performing suspiciousness measures [18,19,40, 71,130,140,149,150,158,161–163,181,194,197,238,241,242,246,254,262]. Accordingly, the contributions of this chapter are as follows:

- *New measures.* We introduce and motivate 95 new measures (borrowed from other areas of science and philosophy) to SBFL. These measures are divided into four categories: similarity, prediction, causal, and confirmation measures. We also automatically generate thousands of new measures.

- *Theoretical component.* In order to identify a class of measures for experimental comparison, we formally prove that many measures are ranking equivalent for the purposes of fault localisation.

- *Experimental component.* To empirically evaluate our new measures we provide what is, to our knowledge, the largest scale experimental comparison in software fault localisation to date, comparing all known measures on over 50k+ program versions. Our results identify a new best performing measure $m9185$ which alongside Kulkzynski2, statistically significantly outperforms all other established measures ($p = 0.01$) at W-scores. We also demonstrate that many of our new measures are competitive.

The rest of this chapter is organised as follows. In Section 5.1 we introduce and motivate our new SBFL measures, and then in Section 5.2 formally prove that large classes of measures are equivalent for the purposes of SBFL. In Section 5.3 we present

our experimental results and discussion, and in Section 5.4 we present our conclusions. This chapter is based on the paper of Landsberg et al. [149].

5.1 New Measures

In this section we introduce and motivate many new measures to SBFL. We first describe our overall strategy, which is based on some general observations about different attitudes to measures in the SBFL literature.

Over the past few years, the divide has become stronger between experimental approaches, and theoretical approaches. We emphasise the distinction as follows. In the theoretical approach, the development of a measure is driven *a priori*. This has involved identifying formal properties for a measure to satisfy (such as strict rationality [179]); many of which are particular to sub-problems in SBFL (such as single fault optimality [181]). In the experimental approach, the development of a measure is driven *a posteriori* – on behalf of performance on fault localisation benchmarks, with measures generated using automated or machine learning methods (e.g. Yoo's genetic algorithm derived measures [254]), or a manually tweaked to best fit the benchmarks (e.g. Wong-III measure [241]).

We argue that both approaches are valuable endeavours in treating the problem of SBFL – informing our general strategy of introducing many different measures with many different sources. The theoretical approach is important in discovering a priori exploitable properties (such as those mentioned above), and the experimental approach is important in discovering new insights in the design space when discovery of more complex formal properties become more difficult (this point is also defended by Yoo. See [254] for discussion). This resonates with what Yoo calls "the mantra of Search Based Software Engineering", namely: It is easier to compare solutions and choose the better one than to design a perfect solution from the scratch.

Accordingly, in this section we contribute to both approaches. For the theoretical approach, we introduce 95 measures new to SBFL but which were developed in other fields to satisfy certain a priori properties. We argue that many of their underlying approaches make them potentially suitable candidates to apply in SBFL. For the experimental approach, we automatically generate thousands of measures which satisfy desirable formal properties discussed in 3.6 with the aim of garnering insight into what works in practice. We argue this combined approach is a good strategy in exploring the space of potentially good SBFL measures – our automated technique

performs a job of covering massive areas of the possible space of very simple measures, standing as the "lower hanging fruit". In contrast, many measures from the literature are probably too complex in structure to easily discover using many types of machine learning methods, and consequently stand as "higher hanging fruit".

The remainder of this section is organised as follows. We first introduce our new suspiciousness measures, organised into five different groups: *similarity*, *prediction*, *causal*, *confirmation*, and *automatically generated* measures (presented in subsections 5.1.1, 5.1.2, 5.1.3, 5.1.4, and 5.1.5 respectively). Their application to SBFL is motivated in terms of different and competing proposals about what a suspiciousness measure should capture, which we discuss at the beginning of each subsection. In Sec. 5.1.5 we also introduce thousands of machine generated measures which satisfy desirable formal properties.

5.1.1 Similarity measures

A first proposal is as follows, and is well established proposal in the literature [179, 181]:

- A suspiciousness measure should measure how *similar* a C is to the error E.

This proposal motivates the use of similarity measures for SBFL. Following the established trend in the literature [163, 181], we likewise introduce many similarity measures which were originally developed in other domains, and propose to put them to new use in SBFL. This general strategy is motivated by Naish et al, who argue that the general behaviour of established similarity measures are what we would desire of an SBFL measure. This is because many similarity measures measure the degree to which two variables are associated, where it is assumed that the higher the association between a given program component and the error, the more likely that component is to be a fault.

Mathematically, the similarity between a component and the error in a program model C_i and E may be symbolised $s(C_i, E)$, and defined in terms of a probabilistic expression as described in section 3.4. More generally, and as discussed by Naish et al [181] in the context of SBFL, a similarity metric s is expected to satisfy the following properties for all $C_i, C_j, C_k \in \boldsymbol{PM}$: positivity ($s(C_i, C_j) > 0$ if $i \neq j$), symmetry ($s(C_i, C_j) = s(C_j, C_i)$), and triangle inequality ($s(C_i, C_j) + s(C_j, C_k) \geqslant s(C_i, C_k)$). Accordingly, new proposed similarity measures are presented in Tables 5.1 and 5.2.

In all our tables, we abbreviate $|\mathbf{T}|$, $|\mathbf{F}|$, $|\mathbf{P}|$ with T, F, P respectively in order to simplify our presentation.

Note that for several measures in the table the measure can equal a negative number (namely, the bottom seven measures of Table 5.1). This violates the positivity condition of similarity measures. Here, the measures were created for experimental purposes by taking established distance measures, and adding a minus sign in front of them in order give us a measure of *negative distance*. We included these measures in our collection of similarity measures for the purposes of our experiments, although strictly speaking they are not similarity measures.

We now overview details of the new measures. We first present the biometrics. 3w-Jaccard is a weighted version of the Jaccard biometric [47]. Nei&Li was originally developed to measure genetic variation [183]. Sokal&Sneath-I, II, III, IV, and V as general measures of biological taxonomy [207, 208], Faith is an asymmetric similarity measure for ecological categorisation [75]. Gower&Legendre is a general coefficient of similarity [100], Bray&Curtis was developed for ecological categorisation in forestry [39], Forbes-I was developed for measuring the associative relations of species [80], Sorgenfrei was developed for clustering molluscs [209], Mountford was developed for zoological classification [177], Otsuka was developed for biogeographical classification [185]. Tarwid was developed for fish taxonomy [220], Braun-Blanquet for plant classification [38], Fager&McGowan was developed for plankton classification [74], Forbes-II was developed for determining the degree the association between species [80]. Gower was developed as a general biometric [99], Cole as a measure interspecific association [50]. Ochiai-II was developed for zoogeographical categorisation of fish neighbourhoods [47], Baroni-Urbani-Buser-I and II were developed as measures of similarity in zoology [33]. As general distance measures, Size-difference, Shape-difference, Pattern-difference, Dispersion and Vari are measures used in many fields (see [47]). Fossum, MCconnaughey, Johnson, Eyraud, Michael and Dennis are measures presented in the same survey.

From the domain of anthropology, Driver&Kroeber was originally developed to measure cultural relationships [66]. Lance&Williams is a similarity measure originally developed for cluster analysis [147], Stiles was designed to measure association in information retrieval Hellinger to measure the similarity of quadratic forms [114], Cosine was developed for data mining, Chord is a variation of Hellinger [47]. F1 is a function of sensitivity and specificity measures [224]. In general, many of our new similarity measures are available from the surveys of [47].

Name	Measure
Nei&Li	$\frac{2a_{ef}}{2a_{ef}+a_{ep}+a_{nf}}$
3w-Jaccard	$\frac{3a_{ef}}{3a_{ef}+a_{ep}+a_{nf}}$
Sokal&Sneath-I	$\frac{aef}{aef+2aep+2anf}$
Sokal&Sneath-II	$\frac{2a_{ef}+2a_{np}}{2a_{ef}+2a_{np}+a_{ep}+a_{nf}}$
Sokal&Sneath-III	$\frac{a_{ef}+a_{np}}{a_{ep}+a_{nf}}$
Faith	$\frac{a_{ef}+0.5a_{np}}{a_{ef}+a_{np}+a_{ep}+a_{nf}}$
Gower&Legendre	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+0.5a_{ep}+0.5a_{nf}}$
Cosine	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}^2}$
Forbes-I	$\frac{Ta_{ef}}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$
Fossum	$\frac{T(a_{ef}-0.5)^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$
Sorgenfrei	$\frac{{a_{ef}}^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$
Mountford	$\frac{a_{ef}}{0.5(a_{ef}a_{ep}+a_{ef}a_{nf})+a_{ep}a_{nf}}$
ShapeSimilarity	$\frac{T(a_{ep}+a_{nf})-(a_{ep}+a_{nf})^2}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$
PatternSimilarity	$-\frac{4(a_{ep}a_{nf})}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$
SizeSimilarity	$-\frac{(a_{ep}+a_{nf})^2}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$
Lance&Williams	$-\frac{(a_{ep}+a_{nf})}{2a_{ef}+a_{ep}+a_{nf}}$
Bray&Curtis	$-\frac{(a_{ep}+a_{nf})}{2a_{ef}+a_{ep}+a_{nf}}$
Vari	$-\frac{(a_{ep}+a_{nf})}{4(a_{ef}+a_{ep}+a_{nf}+a_{np})}$
Hellinger	$-2\sqrt{(1-\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})F}})}$
Chord	$-\sqrt{2(1-\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})F}})}$

Table 5.1: New similarity measures 1/2

Name	Measure
Driver&Kroeber	$\frac{a_{ef}}{2}(\frac{1}{a_{ef}+a_{ep}}+\frac{1}{a_{ef}+a_{nf}})$
Johnson	$\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{a_{ef}+a_{nf}}$
Dennis	$\frac{a_{ef}a_{np}-a_{ep}a_{nf}}{\sqrt{T(a_{ef}+a_{ep})(a_{ef}+a_{nf})}}$
Braun-Blanquet	$\frac{a_{ef}}{\min(a_{ef}+a_{ep},a_{ef}+a_{nf})}$
Dispersion	$\frac{a_{ef}a_{np}-a_{ep}a_{nf}}{(a_{ef}+a_{ep}+a_{nf}+a_{ep})^2}$
Michael	$\frac{4(a_{ef}a_{np}-a_{ep}a_{np})}{(a_{ef}+a_{np})^2+(a_{ep}+a_{np})^2}$
Baroni-Urbani-Buser-I	$\frac{\sqrt{a_{ef}a_{np}}+a_{ef}}{\sqrt{a_{ef}a_{np}}+a_{ef}+a_{np}+a_{nf}}$
Baroni-Urbani-Buser-II	$\frac{\sqrt{a_{ef}a_{np}}+a_{ef}-(a_{ep}+a_{nf})}{\sqrt{a_{ef}a_{np}}+a_{ef}+a_{np}+a_{nf}}$
Peirce	$\frac{a_{ef}a_{np}+a_{ep}a_{nf}}{a_{ef}a_{ep}+2a_{ep}a_{nf}+a_{nf}a_{np}}$
Otsuka	$\frac{a_{ef}}{((a_{ef}+a_{ep})(a_{ef}+a_{nf}))^{0.5}}$
Mcconnaughey	$\frac{{a_{ef}}^2-a_{ep}a_{nf}}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$
Fager&McGowan	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})F}}-\frac{\max(a_{ef}+a_{ep},F)}{2}$
Forbes-II	$\frac{Ta_{ef}-(a_{ef}+a_{ep})F}{Tmin(a_{ef}+a_{ep},F)-(a_{ef}+a_{ep})F}$
Gower	$\frac{a_{ef}+a_{np}}{\sqrt{(a_{ef}+a_{ep})FP(a_{nf}+a_{np})}}$
Sokal&Sneath-IV	$\frac{\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{F}+\frac{a_{np}}{P}+\frac{a_{np}}{a_{nf}+a_{np}}}{4}$
Sokal&Sneath-V	$\frac{a_{ef}a_{np}}{((a_{ef}+a_{ep})FP(a_{nf}+a_{np}))^{0.5}}$
Cole	$\frac{\sqrt{2}(a_{ef}a_{np}-a_{ep}a_{nf})}{\sqrt{(a_{ef}a_{np}-a_{ep}a_{nf})^2-(a_{ef}+a_{ep})FP(a_{nf}+a_{np})}}$
Stiles	$\log\frac{T(\lvert a_{ef}a_{np}-a_{ep}a_{nf}\rvert-\frac{T}{2})^2}{(a_{ef}+a_{ep})FP(a_{nf}+a_{np})}$
Eyraud	$\frac{T^2(Ta_{ef}-(a_{ef}+a_{ep})F)}{(a_{ef}+a_{ep})FP(a_{nf}+a_{np})}$
Fager	$\frac{a_{ef}}{\sqrt{F(a_{ef}+a_{ep})}}-\frac{1}{2\sqrt{F}}$
F1	$\frac{2}{\frac{1}{\frac{aef}{aef+aep}}+\frac{1}{\frac{aef}{aef+anf}}}$
Simpson	$\frac{a_{ef}}{\min(a_{ef}+a_{ep},a_{ef}+a_{nf})}$
Tarwid	$\frac{Ta_{ef}-(a_{ef}+a_{ep})(a_{ef}+a_{nf})}{Ta_{ef}+(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$

Table 5.2: New similarity measures 2/2

5.1.2 Prediction measures

We argue that the established proposal of using symmetric similarity measures for SBFL can be improved upon in terms of motivation. Thus, a second proposal is as follows:

- A suspiciousness measure should measure the degree to which the execution of a component C *predicts* the error E.

This proposal motivates the use of what we loosely call prediction or statistical measures, and to our knowledge is a new proposal. Many measures of prediction can also be found in the surveys of Choi [47] and Tan [219]. Mathematically, many statistical measures differ from similarity measures insofar as they break the condition of symmetry, and were originally used to measure the predictive relationship between a predictor and its effects.

We describe and motivate some of the prominent measures in this group here. In general, many of these measures are commonly used in epidemiology and diagnosis to estimate how well a test result predicts a disease or a successful treatment (for example, see Fletcher [79] and Everitt [73]). Four prominent measures are as follows: PPV (positive predictive value), NPV (negative predictive value), sensitivity (a.k.a. true positive rate, or recall rate) and specificity (a.k.a. true negative rate, or precision rate). These measures are naturally grouped together as their corresponding probabilistic expressions are the elementary conditional probabilities $P(E|C)$, $P(\neg E|\neg C)$, $P(C|E)$, $P(\neg C|\neg E)$ respectively. Applied to SBFL, $P(E|C)$ can be understood to measure how much the execution of a program component C predicts the error E, $P(\neg E|\neg C)$ measures how much the error E depends on the execution of a given component C, $P(C|E)$ measures how often C is executed when an error occurs, and $P(\neg C|\neg E)$ measures how much the execution of C depends on the error E. We believe these are well motivated candidates to measure degree of suspiciousness.

More complex measures in this group are as follows. The Pearson (I-V) metrics measure the degree to which C and E are correlated or co-dependent. In addition, YoudensJ and PositiveLikelyhood are both functions of sensitivity and specificity, and can equally be defined as $\text{Sensitivity} + \text{Specificity} - 1$, and $\text{Sensitivity}/(1 - \text{Specificity})$ respectively [73, 79]. The new prediction measures are presented in Table 5.3. Note a couple of abbreviations have been used: Pearson-I and Pearson&Heron-I are are abbreviated χ^2 and ρ. These abbreviations are used in the definitions of Pearson-II and Pearson-III respectively.

Name	Measure
PPV	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
NPV	$\frac{a_{np}}{a_{np}+a_{nf}}$
Sensitivity	$\frac{a_{ef}}{a_{ef}+a_{nf}}$
Specificity	$\frac{a_{np}}{a_{np}+a_{ep}}$
Youden's-J	$\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{np}}{a_{np}+a_{ep}} - 1$
PosLikelyhood	$\frac{a_{ep}a_{ef}+a_{ef}a_{np}}{a_{ep}a_{ef}+a_{ep}a_{nf}}$
Tetrachoric	$\cos(\frac{\pi}{1+\sqrt{\frac{a_{ef}a_{np}}{a_{ep}a_{nf}}}})$
Relative risk	$\frac{P(E\|C)}{P(E\|\neg C)} = \frac{a_{ef}a_{nf}+a_{ef}a_{np}}{a_{ef}a_{nf}+a_{nf}a_{ep}}$
Z-ratio	$\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{a_{nf}}{a_{nf}+a_{np}}$
Peirce	$\frac{a_{ef}a_{np}+a_{ep}a_{nf}}{a_{ef}a_{ep}+2a_{ep}a_{nf}+a_{nf}a_{np}}$
Pearson-I (χ^2)	$\frac{T(a_{ef}a_{np}-a_{ep}a_{nf})}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})(a_{ep}+a_{np})(a_{nf}+a_{np})}$
Pearson-II	$(\frac{\chi^2}{T+\chi^2})^{\frac{1}{2}}$
Pearson&Heron-I (ρ)	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{ef}+a_{nf})(a_{ep}+a_{np})(a_{nf}+a_{np})}}$
Pearson-III	$(\frac{\rho}{T+\rho})^{\frac{1}{2}}$
Pearson&Heron-II	$\cos(\frac{\pi\sqrt{a_{ep}a_{nf}}}{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}})$
Anderberg-II	$\frac{\sigma-\sigma'}{2T}$
Tanimoto	$\frac{a_{ef}}{a_{ef}+a_{ep}+a_{nf}}$
Mutual-Info	$\frac{P(C,E)\log\frac{P(C,E)}{P(C)P(E)}}{\min(-P(C)\log P(C)-P(E)-\log P(E))}$
Simpson	$\frac{a_{ef}}{\min(a_{ef}+a_{ep}, a_{ef}+a_{nf})}$
Gilbert&Wells	$\log a_{ef} - \log T - \log(\frac{a_{ef}+a_{ep}}{T}) - \log(\frac{a_{ef}+a_{nf}}{T})$

Table 5.3: New prediction measures

5.1.3 Causal measures

An alternative third proposal is as follows:

- A suspiciousness measure should measure the degree by which a component C has the propensity to *cause* the error E.

This motivates the use of measures of causal power/strength to SBFL, and to our knowledge is a new proposal. Such measures are principally found in the domain of philosophy of science [78] and artificial intelligence [189], and many of their formal properties have been shown [78]. The intuition behind many theories of causal power is that causes raise the probability of their effects. Accordingly, most causal measures begin their development by consideration of the following general principle [117]:

- C causes E just in case C raises the probability of E

Following this, in general is it is assumed that the more a cause raises the probability of an effect, the greater propensity it has with respect to causing it [78], and a causal measure should measure this propensity. Four prominent causal measures which describe this intuition directly are Suppes $= P(E|C) - P(E|\neg C)$ [217], Eels $= P(E|C) - P(E)$ [69], Lewis $= \log \frac{P(E|C)}{P(E|\neg C)}$ [154], and Fitelson-I $= \log \frac{P(E|C)}{P(E)}$ [78]. Applied to SBFL, the proposal is that the more the execution of a given component raises the probability of the error, the more likely it is to be the cause of the error, and thus the more suspicious. We can summarise the motivation for causal measures in the context of SBFL with the maxim that *bugs raise the probability of errors*, and thus a suspiciousness measure should measure the degree to which a given component raises the probability of the error as indicative of its faultiness.

We now summarise the remaining causal measures considered in our comparisons. The Good measure [93, 94] is to our knowledge the first purpose designed causal measure, developed in 1961. Since then the philosophy literature has provided a supply of causal measures. Suppes was developed in 1970 [217], Lewis in 1986 [154], Eels in 1991 [69], and Fitelson-I in 2011 [78]. Other measures from the philosophy literature on causation are Fitelson II and III [78], which were designed in order to contrast against other causal measures [78]. Causal measures drawn from areas other than philosophy include the domain of A.I., in which Pearl describes Suppes' measure as a measure of "causal necessity and sufficiency", and also considers four other causal measures [189], which he describes as follows. Pearl-I measures the degree

of "causal necessity", Pearl-II "causal sufficiency", Pearl-III "causal enablement" and Pearl-IV "causal disablement". The Korb-II/III measures are measures of "causal information" [143] and are based on concepts from information theory (Korb-I is a variation of a measure from [142]). Cheng is the only causal theory that comes from the psychology literature [46], and was designed to model causal reasoning. In general, authors of measures use a wide range of (often differing) assumptions to generate their measures. The new causal measures are presented in Table 5.4.

Name	Measure
Suppes	$P(E\|C) - P(E\|\neg C)$
Eels	$P(E\|C) - P(E)$
Lewis	$\log \frac{P(E\|C)}{P(E\|\neg C)}$
Fitelson-I	$\log \frac{P(E\|C)}{P(E)}$
Fitelson-II	$P(E) - P(E\|\neg C)$
Fitelson-III	$\frac{P(E\|C)-P(E)}{P(\neg E)}$
Pearl-I	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|C)}$
Pearl-II	$\frac{P(E\|C)-P(E\|\neg C)}{P(\neg E\|\neg C)}$
Pearl-III	$\frac{P(C)(P(E\|C)-P(E\|\neg C))}{P(E)}$
Pearl-IV	$\frac{P(\neg C)(P(E\|C)-P(E\|\neg C))}{P(\neg E)}$
Korb-I	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|\neg C)}$
Korb-II	$P(E\|C) \log \frac{P(E\|C)}{P(E)}$
Korb-III	$P(C)P(E\|C) \log \frac{P(E\|C)}{P(E)}$
Good	$\log \frac{P(\neg E\|\neg C)}{P(\neg E\|C)}$
Cheng	$\frac{P(E\|C)-P(E\|\neg C)}{1-P(E\|\neg C)}$

Table 5.4: New causal measures

5.1.4 Confirmation measures

An alternative fourth proposal is as follows:

- A suspiciousness measure should measure the degree by which the execution of component is a *hypothesis* C which explains the evidence for the error E.

This motivates the use of measures of confirmation (sometimes called evidential/inductive/confirmation/explanation measures [119]) to the domain of SBFL, and to our knowledge is a new proposal. Such measures were originally developed in the domain of philosophy of science, and many of their formal properties have been shown [70]. The intuition behind many theories of confirmation is that evidence raises the probability of their hypothesis. Accordingly, many confirmation measures begin their development by consideration of the following general principle [119]:

- A hypothesis C confirms the evidence E just in case the evidence raises the probability of the hypothesis.

Following this, it is assumed that the more the evidence raises the probability of a hypothesis, the greater the evidence confirms it [70], and that a confirmation measure should measure this degree. Four prominent confirmation measures which describe this intuition directly are Earman $= P(C|E) - P(C)$ [67], Joyce $= P(C|E) - P(C|\neg E)$ [134], Milne $= \log \frac{P(C|E)}{P(C)}$ [174], and Good-II $= \log \frac{P(C|E)}{P(C|\neg E)}$ [95]. Accordingly, the new proposal for SBFL is that the more the evidence for the error E raises the probability of a given hypothesis (where a hypothesis expresses the proposition that a given program component is executed), the more likely it is to be the correct hypothesis which explains the error E, and thus the more suspicious it should be assumed to be with respect to being a fault.

Note there is a relationship between confirmation measures and the causal measures discussed in the previous section, insofar as with many measures the variables C and E can be swapped around to convert a causal measure into a confirmation measure. Consequently, both causal and confirmation measures share the motivation that the relationship between a fault and the error involves probability raising. In the case of causal measures it is assumed faults raise the probability of errors, and with confirmation measures it is assumed that errors raise the probability of faults.

We now present an overview of the confirmation measures. The new confirmation measures are Joyce [134], Earman [67], Milne [174] Good-II [95], Carnap-I [43, 78], Carnap-II [146], Crupi [53], Rescher [198], Kemeny [137], Popper-I, II and

III [146, 192, 193], Levi [146, 153], Finch-I [77], Gaifman [83], and Rips [199]. Letting H be the hypothesis and E the evidence, various confirmation measures are described as follows. Carnap-I is a "covariance measure" [43, 78]. Carnap-II is a measure of "increase of confirmation of H" [146]. Crupi measures "the impact of a piece of evidence on the credibility of a hypothesis" [53]. Rescher measures the "degree of evidential support that E gives to H" [198]. Kemeny measures "the degree of factual support" that E gives H [137]. Popper-I measures the "degree of corroboration" of H. Popper-II and Popper-III measures the "degree of explanatory power" of H with respect to E [146, 192, 193]. Levi is described as measuring the degree of "epistemic expectation in accepting H on the evidence of E" [146, 153]. Finch-I measures "confirmation power" [77]. Gaifman measures "confirmation rate" [83]. Rips is the only confirmation measure to come from psychology and not philosophy of science, and measures "inductive strength" [199]. Many of the confirmation measures presented are also discussed in the survey of [119]. The new confirmation measures are presented in Table 5.5.

Name	Measure
Earman	$P(C\|E) - P(C)$
Joyce	$P(C\|E) - P(C\|\neg E)$
Milne	$\log \frac{P(C\|E)}{P(C)}$
Good-II	$\log \frac{P(C\|E)}{P(C\|\neg E)}$
Carnap-I	$P(C \cap E) - P(C)P(E)$
Crupi	$\frac{P(C\|E)-P(C)}{P(\neg C)}$ if Earman ≥ 0 $\frac{P(C\|E)-P(C)}{P(C)}$ otherwise
Carnap-II	$P(C)\frac{P(E\|C)-P(E)}{P(E)}$
Rescher	$\frac{P(C)}{P(\neg C)}P(E\|C) - P(E)$
Kemeny	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|C)+P(E\|\neg C)}$
Popper-I	$P(C)P(C\|E)\frac{P(E\|C)-P(E)}{P(E\|C)+P(E)}$
Popper-II	$\frac{P(E\|C)-P(E)}{P(\neg C)P(E\|C)+P(E)}$
Popper-III	$\frac{P(E\|C)-P(E)}{P(E\|C)+P(E)}$
Levi	$P(C)P(\neg C)\frac{P(E\|C)-P(E\|\neg C)}{P(E)}$
Finch-I	$\frac{P(C\|E)-P(C)}{P(C)}$
Gaifman	$\frac{P(\neg C)}{P(\neg C\|E)}$
Rips	$1 - \frac{P(\neg C\|E)}{P(\neg C)}$

Table 5.5: New confirmation measures

5.1.5 Automatically generated measures

A fifth proposal is as follows.

- In the absence of a priori motivated suspiciousness measures, a suspiciousness measure should be generated using automated or machine learning methods.

To build on work in the area of automatically generated suspiciousness measures [254], we designed a simple automated procedure to generating new measures appropriate for SBFL. Our procedure is based on two steps: First, generate every single possible measure up to a given bound of syntactic complexity. Second, prune this space in order to find a subset of measures that satisfy desirable formal properties (namely, strict rationality and ranking in-equivalence).

The motivation for this idea is as follows. Firstly, generating every possible measure up to a given syntactic bound meant that all the measures were relatively syntactically simple. This property was desirable because, along with simplicity being a theoretical virtue of a good measure, the easier it is to garner general insights into the problem of SBFL. Furthermore we were interested in whether simple machine generated measures could outperform much more complex machine generated ones (such as those developed by Yoo [254]). Secondly, we wanted the set to contain measures which were rational (as defined in the preliminaries), because we agree this property should be held by any SBFL measure [179], and also because it allowed us to substantially prune the space of measures thereby making experimental comparison tractable. Finally, we wanted the set to approximate a set of ranking inequivalent measures, because it is pointless in our context to test thousands of measures which are equivalent for the purposes of ranking, and furthermore contributes to the problem of making experimental comparison on 50k+ programs infeasible.

The formal details of the three steps are as follows. At the first step we generated a large class of measures in a given language. We define a set of variables $VAR_1 = \{a_{ef}, a_{nf}, a_{ep}, a_{np}\}$. and a set of operations $OP = \{+, -, \times, /\}$. We inductively define a syntactic complexity measure sc, as follows:

- $sc(x) = 1$ if $x \in VAR_1$,
- $sc(x \text{ op } y) = sc(x) + sc(y)$ for each op $\in OP$.

We then generated the set of measures $\{m|sc(m) \leq 4\}$. Intuitively, this is the set of expressions with a maximum of four variables from VAR_1 using the operations from OP. This provided for over 50k syntactically different measures.

As this number of measures is intractable to test on our benchmarks, our second step included a pruning process. First, we approximated the subset which satisfied the property of rationality (a desirable property), and then kept the subset of this set which were together maximally ranking inequivalent. Instead of formally proving these properties of the measures (which is intractable), we used sets of thousands of random inputs to test whether a measure was rational (reducing the number of measures from 50k to 8k), and then tested the measures on thousands of benchmarks to find a maximally inequivalent subset (reducing the number from 7k to over 2k). This reduction meant that we had a class of over 2k syntactically simple measures approximated to be both rational and ranking in-equivalent.

Finally, we re-ran the above process for three other sets of variables: $VAR_2 = \{P(C|E),\ P(C|\neg E),\ P(\neg E|C), P(E|C)\}$, $VAR_3 = \{P(C \cap E), P(C \cap \neg E), P(\neg C \cap E), P(C \cap \neg E)\}$, $VAR_4 = \{P(E|C),\ P(\neg E|C),\ P(E|\neg C),\ P(\neg E|\neg C)\}$. This brought our total to over 8k SBFL measures. Each measure was given a numerical code to identify it, such that each measure had a name of the form mn, where n is a natural number. For instance we have

$$m9185 = (P(C|E) + P(E|C)) - \frac{P(\neg E|C)}{P(C|E)}$$

This measure can be seen as a modification of the Kulczynski2 measure in much the same way as the Dstar and Zoltar measures are a modification of the Ochiai measure.

Note that this procedure also generated many existing measures, such as Kulczsynski2, Ochiai, and Tarantula, and in general can be used to automatically generate good measures, avoiding the need for manual trial and error. However, many measures were not generated using this approach (such as the chi squared measure). This was because many measures were more syntactically complex than allowed by the algorithm.

This concludes the presentation of new measures.

5.2 Equivalence Proofs

In this section, we extend the work of Naish et al [181] by providing many of the remaining ranking equivalence proofs (50+) for both new and established suspiciousness measures in this paper (excluding the automatically generated ones). Proving ranking equivalences is important for two reasons. Firstly, we can determine a maximal set of non-equivalent measures which allows us to discard the equivalent measures, thereby making experimentation more tractable. Secondly, we can find monotonic simplifications which help identify the important features of suspiciousness measures, which may in turn be used to guide future development. Our large number of equivalence proofs demonstrates that finding a new suspiciousness measure which is different for the purposes of ranking suspicious program components can potentially be quite hard.

We start with outlining our proof method. In the proofs below, we use Lee's lemma [181].

Lemma 1. A measure m is monotonically equivalent to $f \circ m$ if f is a monotonically increasing function.

An intuition for the lemma can be gained with a simple example. Let m be a SBFL measure a^i_{ef}, and let f be the monotonically increasing function with the definition $f(x) = 2x$. Accordingly, a^i_{ef} is monotonically equivalent to $2a^i_{ef}$ by the lemma.

In general, to show that two measures w_1 and w_2 are equivalent, it suffices to find a monotonically increasing function f such that $f \circ w_1 = w_2$, and then apply Lee's Lemma. There then remains the second step of proving that f is indeed a monotonically increasing function, and the third step of proving the aforementioned equality. After f has been found, the general method for the second and third steps are described below.

Step 2. Given f, we must demonstrate that f is monotonically increasing. Together with the identification of relatively simple monotonically increasing functions (such as $m + 1$, 2^m etc) and decreasing functions (such as $1/m$ etc), f can easily be identified as monotonically increasing function using four rules:

1. If f is increasing on m, and g is increasing, then $g \circ f$ is increasing on m.
2. If f is decreasing on m, and g is decreasing, then $g \circ f$ is increasing on m.
3. If f is decreasing on m, and g is increasing, then $g \circ f$ is decreasing on m.

4. If f is increasing on m, and g is decreasing, then $g \circ f$ is decreasing on m.

Example 5.2.1. To illustrate step 2 and the use of these four rules, we show that $\frac{1}{(1/x)+1}$ is monotonically increasing on x. $1/x$ is decreasing on x, and $(y+1)$ is increasing for any y, thus $(1/x)+1$ is decreasing (by 3). Given $(1/x)+1$ is decreasing on x, and $1/y$ is decreasing for any y, we have the result that $\frac{1}{(1/x)+1}$ is increasing (by 2).

Step 3. We must demonstrate $f \circ w_1 = w_2$ when f is found to be monotonically increasing on x. This can be demonstrated algebraically.

Example 5.2.2. To illustrate step 3, we can trivially show $\frac{1}{(1/x)+1} = \frac{a_{ef}}{a_{ep}+a_{ef}}$ when $x = \frac{a_{ef}}{a_{ep}}$. Thus, the example for the last two steps together with Lee's Lemma is sufficient to show that $\frac{a_{ef}}{a_{ep}}$ is ranking equivalent to the PPV measure $\frac{a_{ef}}{a_{ep}+a_{ef}}$.

Given the simplicity of steps 2 and 3, we omit their explicit inclusion in our proofs, and find it sufficient to provide step 1 which is to show that a measure is ranking equivalent to another by simply providing the monotonically increasing function f that transforms one function into another. This is an accepted convention established by Naish [181]. In many cases, we have tried to find a simplified monotonically equivalent expression for a given a measure (which we call a *redux*).

To aid in the proofs, we use Tables 5.6 and 5.7. *Eq.* denotes groups of equivalent measures (1, 2, 3 etc.), *Name* gives the name of the measure, m is the measure, and f gives a function which when composed with m, evaluates to a measure higher up in the equivalence class.

Abbreviations in the table are as follows: ρ is equal to Geometric Mean, Pearson-I is the χ^2 test $\frac{T(a_{ef}a_{np}-a_{ep}a_{nf})}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})(a_{ep}+a_{np})(a_{nf}+a_{np})}$. See Naish for similar proofs [181]. For simplicity of presentation, T, P, and F abbreviate $|\mathbf{T}|$, $|\mathbf{P}|$, and $|\mathbf{F}|$ respectively.

We now prove that the classes in Tables 5.6 and 5.7 are equivalence classes as follows.

Eq	Name	m	f
1	Redux	$\frac{a_{ef}}{a_{ep}}$	m
	PPV	$\frac{a_{ef}}{a_{ef}+a_{ep}}$	-
	Qe	-	-
	Eels	$P(E\|C) - P(E)$	$m+\frac{F}{T}$
	Fitelson-I	$\log \frac{P(E\|C)}{P(E)}$	$2^m \times \frac{F}{T}$
	Fitelson-III	$\frac{P(E\|C)-P(E)}{P(\neg E)}$	$m\frac{P}{T} + \frac{F}{T}$
	Pearl-IV	-	-
	Cosine	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}^2}$	mF
	Tarwid	$\frac{Ta_{ef}-(a_{ef}+a_{ep})(a_{ef}+a_{nf})}{Ta_{ef}+(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$	$\frac{F(m+1)}{T(1-m)}$
	Fossum	$\frac{T(a_{ef}-0.5)^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$	$\frac{mF}{T}$
	Popper-III	$\frac{P(E\|C)-P(E)}{P(E\|C)+P(E)}$	$m\frac{F}{T}$
2	RelativeRisk	$\frac{P(E\|C)}{P(E\|\neg C)}$	m
	Lewis	$\log \frac{P(E\|C)}{P(E\|\neg C)}$	2^m
	Korb-I	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|\neg C)}$	$m+1$
	Pearl-I	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|C)}$	$\frac{1}{(1/m)-1}+1$
	Kemeny	$\frac{P(E\|C)-P(E\|\neg C)}{P(E\|C)+P(E\|\neg C)}$	$\frac{-2}{m-1}-1$
3	Redux	$\frac{a_{np}}{a_{nf}}$	m
	NPV	$\frac{a_{np}}{a_{np}+a_{nf}}$	$\frac{1}{(1/m)-1}$
	Fitelson-II	$P(E) - P(E\|\neg C)$	$(m-\frac{F}{T})+1$
	Pearl-III	$\frac{P(C)(P(E\|C)-P(E\|\neg C))}{P(E)}$	$(m\frac{F}{T})+\frac{P}{T}$
	Gaifman	$\frac{P(\neg C)}{P(\neg C\|E)}$	$(\frac{m}{F}\cdot T)-1$
	Rips	$1-\frac{P(\neg C\|E)}{P(\neg C)}$	$(\frac{1/(1-m)}{F}\cdot T)-1$
4	Wong-I	a_{ef}	m
	Sensitivity	$\frac{a_{ef}}{a_{ef}+a_{nf}}$	mF
	Support	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	mT
5	OddsRatio	$\frac{a_{ef}a_{np}}{a_{ep}a_{nf}}$	m
	YulesQ	$\frac{a_{ef}a_{np}-a_{ep}a_{nf}}{a_{ef}a_{np}+a_{ep}a_{nf}}$	$\frac{1}{(2/(m+1))-1}$
	Tetrachoric	$\cos(\frac{\pi}{1+\sqrt{\frac{a_{ef}a_{np}}{a_{ep}a_{nf}}}})$	$\cos(\frac{\pi}{\sqrt{x}+1})$
	Pearson-Heron-II	$\cos(\frac{\pi\sqrt{a_{ep}a_{nf}}}{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}})$	see proof

Table 5.6: Monotonic compositions part 1/2

Eq	Name	m	f
6	Kulczynski-I	$\frac{a_{ef}}{a_{nf}+a_{ep}}$	m
	3w-Jaccard	$\frac{3a_{ef}}{3a_{ef}+a_{ep}+a_{nf}}$	$\frac{1}{(3/m)-3}$
	SokalSneath-I	$\frac{aef}{aef+2aep+2anf}$	$\frac{2}{(1/m)-1}$
	Tanimoto	$\frac{a_{ef}}{a_{ef}+a_{ep}+a_{nf}}$	$\frac{1}{(1/m)-1}$
	Lance&Williams	$-\frac{a_{ep}+a_{nf}}{2a_{ef}+a_{ep}+a_{nf}}$	$-\frac{m+1}{2m}$
	Nei&Lei	$\frac{2a_{ef}}{2a_{ef}+a_{ep}+a_{nf}}$	$\frac{1}{(2/m)-2}$
	F1	$\frac{2}{\frac{1}{\frac{aef}{aef+aep}}+\frac{1}{\frac{aef}{aef+anf}}}$	$\frac{1}{(2/m)-2}$
7	Wong-II	$a_{ef}-a_{ep}$	m
	SokalSneath-II	$\frac{2a_{ef}+2a_{np}}{2a_{ef}+2a_{np}+a_{ep}+a_{nf}}$	$\frac{1}{(2/x)-1}\times T-P$
	Gower&Legendre	-	-
	SokSneath-III	$\frac{a_{ef}+a_{np}}{a_{ep}+a_{nf}}$	$\frac{1}{(1/x)+1}\times T-P$
	Vari	$\frac{(a_{ep}+a_{nf})}{4(a_{ef}+a_{ep}+a_{nf}+a_{np})}$	$(x4T)+F$
	SizeSimilarity	$-\frac{(a_{ep}+a_{nf})^2}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$	$\sqrt{x\times T^2}+F$
8	Redux	$P(E\|C)+P(C\|E)$	m
	Johnson	$\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}$	m
	Kulczynski-II	$\frac{1}{2}(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}})$	$m/2$
	MCconnaughey	$\frac{{a_{ef}}^2-a_{ep}a_{nf}}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$	m
9	GeometricMean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$	m
	Pearson-Heron-I	-	-
	Pearson-III	$(\frac{\rho}{T+\rho})^{\frac{1}{2}}$	$\frac{1}{(1/m^2)-1}\times T$
10	Redux	$\frac{a_{ef}a_{np}-a_{ep}a_{nf}}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})(a_{ep}+a_{np})(a_{nf}+a_{np})}$	m
	Pearson-I	χ^2	$\frac{m}{T}$
	Pearson-II	$(\frac{\chi^2}{T+\chi^2})^{\frac{1}{2}}$	$\frac{1}{(1/m^2)-1}\times T$
	Eyraud	$\frac{T^2(a_{ef}a_{np}-a_{ep}a_{nf})}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})(a_{ep}+a_{np})(a_{nf}+a_{np})}$	$\frac{m}{T^2}$
11	Redux	$P(E\|C)P(C\|E)$	m
	Ochiai	$\frac{a_{ef}^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$	m
	Otsuka	-	-
	Hellinger	$-2\sqrt{(1-\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})F}})}$	$-2\sqrt{1-m}$
	Chord	$-\sqrt{2(1-\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})F}})}$	$-\sqrt{2(1-m)}$
	Sorgenfrei	$\frac{{a_{ef}}^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$	$\sqrt{m}$
	Fager	$\frac{a_{ef}}{\sqrt{F(a_{ef}+a_{ep})}}-\frac{1}{2\sqrt{F}}$	$m+\frac{1}{2\sqrt{F}}$
12	Redux	$-a_{ep}a_{nf}$	m
	PatternSimilarity	$-\frac{4(a_{ep}a_{nf})}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$	$\frac{mT^2}{4}$
13	Simpson	$\frac{a_{ef}}{\min(a_{ef}+a_{ep},a_{ef}+a_{nf})}$	m
	Confidence	$\max(\frac{a_{ef}}{a_{ef}+a_{ep}},\frac{a_{ef}}{a_{ef}+a_{nf}})$	see proof

Table 5.7: Monotonic compositions part 2/2

Proposition 1. Classes 1 to 13 in Tables 5.6 and 5.7 are ranking equivalence classes.

Proof. The proposition follows directly from using the details of Tables 5.6 and 5.7. To show that a measure is ranking equivalent to some member higher up in the same class, one can take the measure m, insert it into the function f, and simplify. Some proofs were too complicated to be included in the table, so we have included them as follows.

We do the proof Pearson-Heron-II here (Eq 10). For Pearson-Heron-II, we first note that $\cos(x)$ is monotonically decreasing when $0 \leq x \leq \pi$. Thus, the expression $-\frac{\pi\sqrt{a_{ep}a_{nf}}}{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}}$ is monotonically equivalent to PearsonHeron-II, by substitution of order preserving functions. We then apply $\frac{1}{((1/x)-1)}+1$ to get $\frac{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}}{\pi\sqrt{a_{ep}a_{nf}}}$ We then apply $(\pi x-1)^2$ to get Odds-ratio.

We do the proof for Confidence here (Eq 13). Simpson is $\frac{a_{ef}}{\min(a_{ef}+a_{ep},a_{ef}+a_{nf})}$. This can be expressed as the condition if $a_{ef}+a_{ep} < a_{ef}+a_{nf}$ then $\frac{a_{ef}}{a_{ef}+a_{ep}}$, else $\frac{a_{ef}}{a_{ef}+a_{nf}}$. This is equivalent to the condition if $\frac{a_{ef}}{a_{ef}+a_{ep}} > \frac{a_{ef}}{a_{ef}+a_{nf}}$ then $\frac{a_{ef}}{a_{ef}+a_{ep}}$, else $\frac{a_{ef}}{a_{ef}+a_{nf}}$ (given $x < y$ if and only if $\frac{z}{x} > \frac{z}{y}$). This can be expressed as $\max(\frac{a_{ef}}{a_{ef}+a_{ep}}, \frac{a_{ef}}{a_{ef}+a_{ef}})$, which is equal to Confidence.

□

We now discuss proof verification. The proofs can be practically verified by appeal to automated reasoning programs such as WolframAlpha [10]. To do so, one can substitute the variable m in the f column with the respective measure given in the m column, and enter it into WolframAlpha. For example, taking the Cosine expression $m = \frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}^2}$ and the function $f = mF$, by substitution we have $\frac{a_{ef}}{\sqrt{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}^2}(a_{ef}+a_{nf})$ (given $F = a_{ef}+a_{nf}$). This simplifies to $\frac{a_{ef}}{a_{ef}+a_{ep}}$ which is the PPV measure. Thus Cosine is in equivalence class 1. To our knowledge no appropriate automated methods are available for the task of finding an appropriate f, so this was done manually. Following the strategy of Naish et al [181], two measures were suspected to be monotonically equivalent when they always gave the same ranking results in experimentation.

We also discovered the following trivial equalities, not represented by the table. Joyce and Ample-II are equal. Youden's J + 1 is equal to both. Rogot-II and Sokal&Sneath-IV are equal. Driver&Kroebner and Kulczynski-II are equal. Piatetsky-Shapiro is equal to Carnap-I. Wong-III and Wong-III' are equal if $a_{ep} + a_{ef} > 0$. PPV and Tarantula are equal. Finally, YulesY has been established as monotonically equivalent to YulesQ (see p.434 of Heinz [113]).

An observation is that many diverse and complex measures are ranking equivalent to much simpler ones. Five classes of measures which had simple reduxes were classes 1, 3, 7, 8, and 11 which were equivalent to $\frac{a_{ef}}{a_{ep}}$, $\frac{a_{np}}{a_{nf}}$, $a_{ef} - a_{ep}$, $P(E|C)P(C|E)$, and $P(E|C) + P(C|E)$ respectively.

In summary, the introduction of new measures in the previous section combined with our ranking equivalence proofs, provides for an overall effort allows us to identify many new different classes of well motivated measures for application to SBFL.

5.3 Empirical Evaluation

In this section we present our experimental results. The purpose of the experiment was to find the best overall performing SBFL measures at the task of finding a single fault in a program. The experimental setup for this task is described in full in Chapter 4. The measures compared include all the established and new measures discussed in this thesis. This includes the 110 established measures (see Appendix A for their definitions), the 95 new similarity, prediction, causal, and confirmation measures presented in this chapter, along with the (2k+) automatically generated measures discussed in section 5.1.5. We also created single bug optimised versions of all non-automatically generated measures (205 of them), using the single bug optimiser presented in section 3.6. This provided for a comparison of many measures.

As is convention, to avoid divisions by zero in our measures we added a small constant (0.5) to each of the elements in the program spectrum [181]. Experimentation in the value of this constant did not make significant changes in results.

The rest of this section is organised as follows. We first present the top ten performing measures in section 6.7.1 along with overall averages of the techniques compared. We then present scores for a range of methods as a function of the number of faults introduced into the program in Section 5.3.2. Finally, we discuss these results in section 5.3.3.

5.3.1 Scores by average

To provide a summary of our top results, the measures which received the best overall AVG W-scores are presented in Table 5.8. The first column gives the name of the measure, the second/fifth columns give the AVG W/A scores respectively for all 100 scores. The other columns give the AVG W/A scores for the SIR and Stiemann

Measure	AVG W	SIR W	STI W	AVG A	SIR A	STI A
m9185	**4.93** (1)	**8.40** (1)	2.63 (2)	**47.39** (1)	**38.15** (1)	53.55 (16)
Kulczynski2	5.12	8.76	2.69	45.09	33.75	52.65
Zoltar	6.09	11.34	2.59	43.26	28.10	53.37
Opt-Ochiai	6.38	10.95	3.34	37.75	21.80	48.38
D3	6.38	10.87	3.39	40.62	27.52	49.35
Ochiai	6.58	11.99	2.97	43.34	29.65	52.47
PattSim	6.62	11.24	3.54	43.84	34.17	50.30
D2	6.77	12.08	3.24	41.86	28.65	50.67
Popper2	6.82	12.30	3.17	43.27	29.90	52.18
Relative-Risk (aka **Lewis**)	6.85	12.48	3.10	38.72	21.73	50.05

Table 5.8: SFL task – top 10 W scorers

(STI) benchmarks respectively. Where our top performer (m9185) ranks for a given scoring method is given in brackets. For the complete tables of AVG W and A scores see Tables 5.9 and 5.10 below. In Appendix A we provide tables which describe the 100 scores for individual benchmarks for a range measures for the interested reader.

For the purposes of clarity, we have only included the top performing new automatically generated measure and the top performing single bug optimised measure in the table 5.8. Thus, where we experimented with 2K+ automatically generated measures and 200 single bug optimised measures, we present the best performer of each class. The best performing single fault optimal measure was $Opt(Ochiai)$, the best performing automatically generated measure was $m9185$. In general many $Opt(g)$/automatically generated measures performed well – we include the top handful of each category in our larger tables in sections 5.10 and 5.9 to illustrate. Also for the purposes of clarity – in all our tables of results, measures proved equivalent (or had equivalent results in our experiments for each of our 60k+ experiments) are represented by one measure per equivalence class (with preference given to measures already established in SBFL). New measures which are inequivalent to established measures are presented in **bold**.

In the remainder of this section we present the AVG W-scores and A-scores for the compared measures in tables 5.9 and 5.10. We also present Figure 5.1, which is constructed as follows. First, for each n-fault benchmark (of which there are 100), the following procedure was performed for a measure. If $y\%$ of the program versions in that n-bug benchmark have a W-score $\leq x\%$, a point is plotted on the graph at (x, y). To create the mean plot, we averaged each of the 100 x's for each y for each of the 100 n-bug benchmarks.

Measure	W-Score	Measure	W-Score	Measure	W-Score
m9185	4.93	Jaccard	8.15	Tarantula	9.53
m30180	5.04	GP22	8.18	Levi	9.62
Kulczynski2	5.12	**Earman**	8.19	GP19	9.66
m3156	5.13	Naish	8.20	GP07	9.70
m16	5.13	GP13	8.20	GP06	9.70
m9830	5.54	GP20	8.20	OneWaySupport	9.81
Zoltar	6.09	GP11	8.20	Stiles	9.87
Opt-Ochiai	6.38	GP26	8.21	Zhang	9.99
D3	6.38	GP29	8.23	Finch1	9.99
Opt-K2	6.49	GP10	8.28	GP16	9.99
Opt-Lewis	6.52	GP15	8.28	Interest	9.99
Ochiai	6.58	**PearlII**	8.29	**Good2**	9.99
PattSim	6.62	**Good**	8.29	OddMultiplier	9.99
D2	6.77	**Cheng**	8.29	**PosLikelyhood**	9.99
Popper2	6.82	AddedValue	8.33	**Keynes**	9.99
RR/**Lewis**	6.85	calb	8.33	Forbes1	9.99
M2	7.10	cald	8.33	Korb2	9.99
YulesQ	7.19	GP23	8.33	GilbertWells	9.99
Conviction	7.20	CBISqrt	8.45	ExampleandCounter	9.99
Mountford	7.35	cale	8.45	SebagSchoenauer	9.99
NPV	7.53	calf	8.45	GP27	10.31
Certainty	7.64	GP25	8.48	Scott	10.47
Pearson1	7.68	**Cohen**	8.52	Peirce	10.54
Suppes	7.68	GP28	8.57	**Rescher**	10.57
Phi	7.72	Kappa	8.59	Fleiss	10.78
GeometricMean	7.76	GP09	8.67	GP03	10.88
GP08	7.76	GP05	8.68	Faith	11.01
Klosgen	7.76	cala	8.81	Laplace	11.12
AMean	7.79	calc	8.83	Confidence	11.16
HMean	7.80	CBIlog	8.86	LeastContradiction	11.44
SokalSneath4	7.80	Ochiai2	8.89	WongII	11.46
GP12	7.86	**SokalSneath5**	8.89	Leverage	11.81
Dennis	7.95	Popper1	8.94	GoodKruskal	12.80
Crupi	7.95	GP21	8.94	Anderberg2	12.87
TwoWaySupport	8.03	GP01	8.94	WongI	12.89
WongIII	8.03	GP14	9.03	M1	12.97
Carnap1	8.05	GP24	9.03	GP02	14.39
Korb3	8.05	InfoGain	9.07	FagerMc	14.60
Dispersion	8.11	JMeasure	9.14	GP04	15.36
YoudensJ	8.11	Michael	9.17	Gower	17.83
Ample2	8.11	GP30	9.19	BinaryNaish	17.89
GP18	8.12	**Carnap2**	9.50	**Random**	23.88

Table 5.9: Overall AVG W-scores

Measure	A-Score	Measure	A-Score	Measure	A-Score
m9185	47.39	Ample2	42.63	GP12	36.56
Klosgen	47.11	**YoudensJ**	42.61	M2	36.53
YulesQ	45.91	Dispersion	42.61	GP16	35.85
m30180	45.77	OneWaySupport	42.32	**Crupi**	35.59
cala	45.52	Jaccard	42.18	GP27	35.31
AddedValue	45.51	D2	41.86	**NPV**	34.84
Certainty	45.29	LeastContradiction	41.76	Leverage	34.75
calc	45.22	Fleiss	41.71	GP30	34.14
m16	45.17	**m9830**	41.38	GP24	33.99
Kulczynski2	45.09	WongII	41.28	GP23	33.99
m3156	45.08	**Cheng**	41.19	GP14	33.99
TwoWaySupport	44.65	**PearlII**	41.19	WongIII	33.93
Korb3	44.63	**Good**	41.19	GP01	33.77
SokalSneath4	44.58	InfoGain	41.19	GP21	33.69
HMean	44.58	D3	40.62	GP19	33.24
CBIlog	44.49	Faith	40.62	GP18	32.46
Phi	44.38	Stiles	40.57	GP08	32.22
GeometricMean	44.36	**Popper1**	40.54	GP28	32.17
calb	44.25	**PosLikelyhood**	40.37	GP22	31.93
cald	44.25	Finch1	40.36	GP15	31.92
AMean	44.17	OddMultiplier	40.33	GP26	31.72
Ochiai2	43.97	**Keynes**	40.33	Naish	31.70
SokalSneath5	43.97	Interest	40.33	GP20	31.70
PattSim	43.84	GilbertWells	40.32	GP11	31.70
calf	43.72	Zhang	40.31	GP13	31.70
cale	43.72	ExampleandCounter	40.28	GP29	31.61
JMeasure	43.71	Good2	40.28	GP10	31.55
CBISqrt	43.59	SebagSchoenauer	40.28	GP09	30.71
Mountford	43.54	Forbes1	40.26	GP25	29.30
Dennis	43.52	**Korb2**	40.26	GP03	28.96
Conviction	43.50	GP06	40.25	**Rescher**	26.13
Cohen	43.40	GoodKruskal	39.58	M1	26.06
Kappa	43.37	Peirce	39.52	GP07	25.60
Ochiai	43.34	RR/**Lewis**	38.72	GP05	22.55
Popper2	43.27	FagerMc	38.51	Laplace	20.83
Pearson1	43.26	**Carnap2**	38.18	Confidence	20.74
Suppes	43.26	Levi	38.17	GP02	20.27
Zoltar	43.26	Opt-K2	37.90	GP04	14.84
Scott	43.22	Opt-Ochiai	37.75	Gower	11.24
Tarantula	43.08	Michael	37.72	**Random**	10.45
Earman	42.74	Opt-Lewis	37.05	WongI	10.01
Carnap1	42.65	Anderberg2	36.60	BinaryNaish	1.99

Table 5.10: Overall AVG A-scores

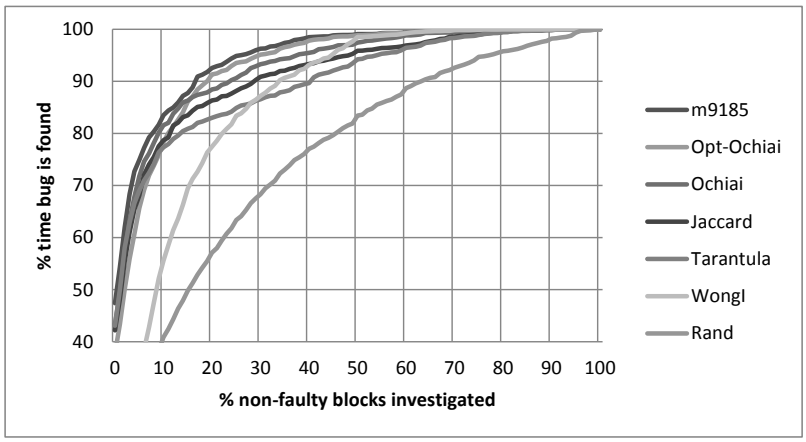

Figure 5.1: Performance of some prominent measures

5.3.2 Scores by number of faults

In Figures 5.2 and 5.3 we present some detailed W-score results for a range of measures. The figures are constructed as follows. On the x axis we give the name of the measure, on the y axis the average W-score, where the key colour shade represents the number of faults. The W-score for each column was calculated by taking the average of all the W-scores for the n-fault versions in that set of benchmarks for that measure. For example, the $m9185$ measure received an average W-score of just over 11% for the 1-fault versions of the SIR benchmarks, where this average was an average of the 10 W-scores computed over each of 1-fault benchmark versions (i.e. the 1 fault `gzip` versions, the 1 fault `tcas` versions etc.). This presentation provides a description of how techniques behave when the number of faults in a program increases. Figures 5.4 and 5.5 are constructed in the same way as Figures 5.2 and 5.3, except the results report A-scores instead of W-scores. We observe that the results in Figures 5.4 and 5.2 are robust – each of the columns are each a function of 10,000 individual experiments.

In these figures we have only chosen a select group of measures, in order to give an impression of the variation in performance and general trends of the SBFL framework. We include $m9185$ because it is our top performing measure. We include Ochiai, Jaccard and Tarantula because of their prominence in the literature. We also include the best performing single bug optimised measure $Opt(Ochiai)$ because of its contrasting behaviour and as an example of a single bug optimised measure.

5.3.3 Discussion

In this section we discuss our experimental results. In general, our discussion is centred around ascertaining trends in behaviour of the SBFL approach in general. We begin as follows:

We first ask, what was the most effective measure in our experiments? The most effective measure is identified with the measure which received the best AVG A and W scores. This was $m9185$, which received the best AVG W-score (4.93) and also the best AVG W-score (47.39). We now discuss significance results. $m9185$ and Kulkzynski2 performed significantly better than all other measures at W-scores using $p = 0.01$. $m9185$ and Klosgen performed significantly better than all other measures at A-scores using $p = 0.01$. For simplicity, the significance result concerning W-scores

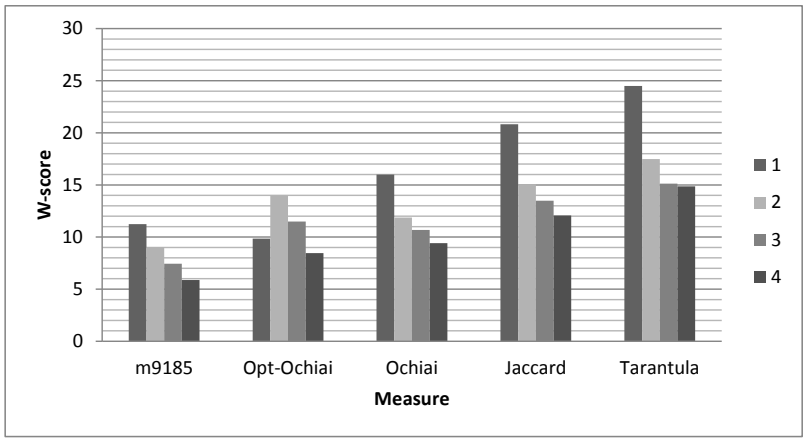

Figure 5.2: W-scores for selected measures on SIR Benchmarks

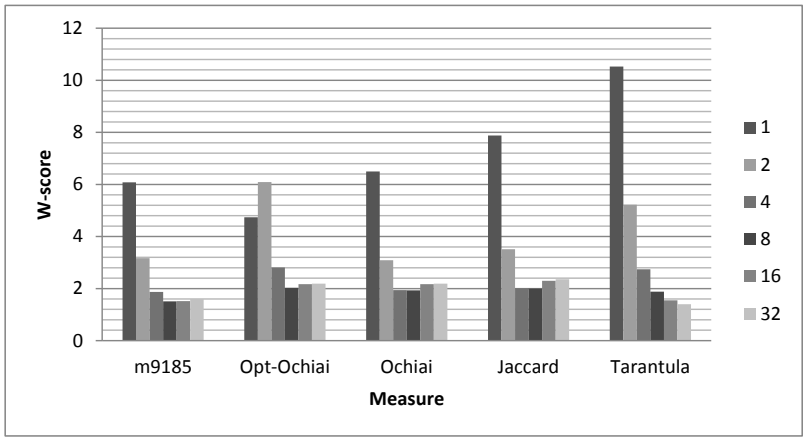

Figure 5.3: W-scores for selected measures on Steimann Benchmarks

excludes the other new automatically generated measures introduced in section 5.1.5. If we include all new automatically generated measures in the comparison, then we have the result that no measure was statistically significantly better than the W-scores of our top five automatically generated measures and Kulkzynski2 (and none in this

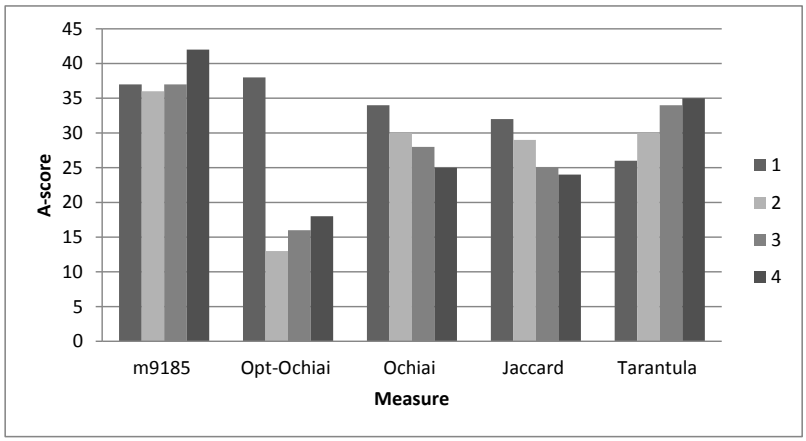

Figure 5.4: A-scores for selected measures on SIR Benchmarks

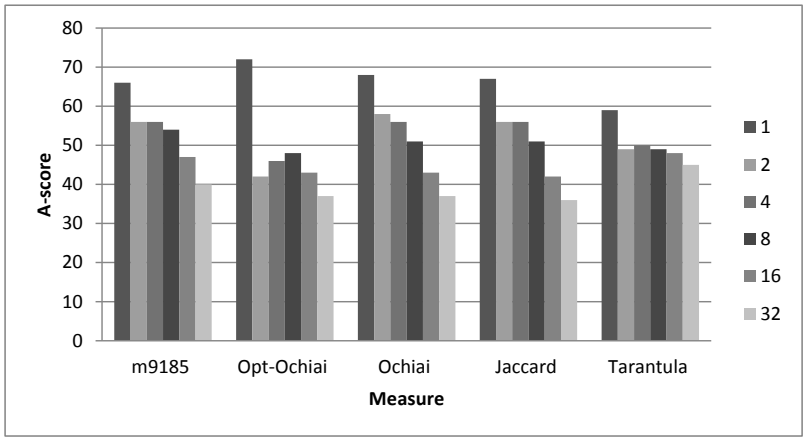

Figure 5.5: A-scores for selected measures on Steimann Benchmarks

class was significantly better than any other in that class). Thus, the experiment identifies $m9185$ as the most effective measure in our experiments.

We now discuss the difference in effectiveness between the best and a worst performing measure. We identify $m9185$ as the best performing measure and identify

$GP04$ as an example of a worst performing measure. The former is ranked 1st in Tables 5.9 and 5.10, and is therefore the best performer in terms of AVG W and A scores (receiving an AVG W score of 4.93 and AVG A score of 47.39). In contrast, $GP04$ is ranked 4th and 5th last in the respective tables, and is therefore one of the worst performers in terms of AVG W and A scores (receiving an AVG W score of 15.36 and AVG A score of 14.84). To summarise the difference between $m9185$ and $GP04$, with respect to W-scores, one would expect to investigate over 3 times more blocks of code in order to find a single fault if one used $GP04$ as opposed to $m9185$. Likewise, with respect to A-scores, one would expect to find a single fault immediately 3 times more often comparing the same measures. In general, this suggests that choice in measure can make a substantial difference in overall fault localisation effectiveness.

We now discuss how pre-established manually generated measures perform overall. Many prominent pre-established measures appear quite low on the tables of overall A and W scores, such as Jaccard and Tarantula. However, many established measures perform well such as Kulkzynski2, Zoltar, Ochiai, the Dstar measures, and single bug optimal measures $Opt-g$. Thus, our larger-scale comparison accentuates the successes and failures of established measures. In general, the relative performance of different measures is confirmed by previous studies [18, 19, 40, 71, 130, 140, 149, 150, 158, 161–163, 181, 194, 197, 238, 241, 242, 246, 254, 262]. In particular, Ochiai and Kulkzynski2 have been consistently confirmed to be top performing measures at W scores in many of these studies. We emphasise that, as been reported before, small differences in experimental setup for our SIR benchmarks can make changes in the absolute values of the scores reported [181], despite the relative performance remaining similar.

We now discuss how the new similarity, predictive, confirmation, and causal measures perform. A measure from each of these classes was shown to be competitive in terms of their AVG W-scores. The best new similarity measure was Pattern similarity, the best new new predictive measure was NPV (Negative predictive value), the best new causal measure was Lewis (which we proved to be ranking equivalent to relative risk = RR), and the best new confirmation measure was Popper2. We discuss these measures as follows. Each one of these appear in the Table 5.8 (the exception being NPV which is a close runner up to Lewis).

We first discuss our top performing new predictive measure NPV = $P(\neg E|\neg C)$. It is noticeable that NPV performed a lot better than PPV = $P(E|C)$, meaning that negative predictive value is more important than positive predictive value, suggesting probabilistic necessity is more important than probabilistic sufficiency when measuring suspiciousness. We add that the second best performing new predictive

measure was Pearson-I, aka the chi-squared test of significance, suggesting that measures which break the conventional SBFL assumption that the distribution is unknown can perform well.

We second discuss our top performing new similarity measure Pattern-similarity $= \frac{-4a_{ep}a_{nf}}{(a_{ef}+a_{nf}+a_{np}+a_{ep})^2}$ here. We first note that our ranking equivalence proofs found Pattern-similarity to be equivalent to the following simpler measure: $-a_{ep}a_{nf}$. This measure came sixth in our top ten table, and is therefore highly competitive. This result dismantles any myths about measures needing to be complex in order to be effective. Indeed, the Wong-III measure is significantly more complex and performs worse than Pattsim in both tables of A and W scores.

We third discuss our top performing new confirmation measure Popper2 $= \frac{P(E|C)-P(E)}{P(\neg C)P(E|C)+P(E)}$. Popper2 is something of an outlier wrt to the performance of confirmation measures. Other than Popper2, confirmation measures (which are designed around the intuition that the errors raised the probability of bugs) in general performed less well than causal measures (which were designed around the converse intuition that bugs raise the probability of errors).

We finally discuss our top performing new causal measures. In section 5.1.3 we motivated and introduced four measures which described the intuition that bugs raise the probability of errors. Their performance are as follows. The performance of Lewis $= \frac{P(E|C)}{P(E|\neg C)}$ (9th in our top ten table) and Suppes $= \frac{P(E|C)}{P(E|\neg C)}$ (following closely behind Lewis in the Table of W scores) experimentally confirms the utility of the proposal that bugs raise the probability of errors, outlined in section 5.1.3. Our monotonicity proofs showed that the remaining two causal measures $\frac{P(E|C)}{P(E)}$ and $P(E|C) - P(E)$ are ranking equivalent to Tarantula, and consequently performed less well.

We now discuss how automatically generated measures performed. We discuss the performance of two different classes of automatically generated measures, Yoo's genetic measures and then our own. We first discuss how the established automatically generated measures generated by Yoo's genetic algorithm (GP_n for $0 \leq n \leq 30$). We observed that their performance clustered (Tables 5.9 and 5.10, and performed a lot less well than simple established measures that were not trained on benchmarks (e.g. Kulkzynski2 requires the inspection of almost 25% less code, measured in terms of AVG W score, than the best genetic measure GP08) (Similar observations hold for the combination measures of Kim; cal_a, cal_b etc.). This suggests that many machine learned measures may suffer from being over-fitted to the original data they were trained on, and are consequently unable to work as well when used on a new data set (such as the Steimann dataset).

We now discuss our new automatically generated measures. For the purposes of clarity we have only included our top performing new automatically generated measures in our tables. However, in general we discovered that over 300 different rational automatically generated measures outperformed Tarantula in terms of AVG W scores, of which $m9185 = (P(C|E) + P(E|C))$ - $\frac{P(\neg E|C)}{P(C|E)}$ is our best representative (with $m30180 = (P(E|C)P(C|E) + P(E|C)^2) + P(C|E)$ following closely in Table 5.9). $m9185$ can be understood as a modified version of the Kulczynski2 measure which preserves rationality in much the same way as the Zoltar and D3 measures are modified versions of Jaccard and Kulczynski measures respectively.

However, two theoretical criticisms of this measure are as follows. Given that our current experiment confirms the suspicion that automatically generated measures can suffer from over-fitting (namely the GP measures), $m9185$ can potentially equally be charged with the same criticism. Secondly, the intuitive connection to SBFL is not entirely obvious. Finally, of our languages for VAR, those measures from language (2) scored highest. The remaining did not score well and therefore have not been reported.

We now discuss some tendencies concerning the relationship between a measure's A and W scores. We observe that in general many measures with good AVG W scores also have good AVG A scores. Kulkzynski2 and $m9185$ are two of the most prominent examples, with $m9185$ maintaining its top position in both cases in Tables of AVG A and AVG W-scores. However, in general there was no strict correspondence between a good AVG A and a good AVG W scores – some measures with relatively bad W-scores scores have good A scores (such as Klosgen), and some measures which have relatively bad A scores have good W scores (such as O_{K2}). In summary, there was not a strict relationship between scoring well with AVG A scores as with AVG W scores.

We now discuss how does limiting oneself to investigating a certain percentage of the program affects fault localisation effectiveness. To discuss this, we appeal to the performance plot of Figure 5.1, which gives us a representation of how effective measures are if we limit ourselves to investigating n percent of a given program. The general trend is as follows: effectiveness grows logarithmically as the investigation into the program continues. In general, measures achieve most of their fault localisations in the first 10% of the blocks investigated (accounting for the steep incline), and that when the search continues after the first 10% the user gets diminishing returns. This confirms our results in [149].

We now discuss differences between measures. First, $m9185$ consistently outperforms all other techniques regardless of the value of n - we could not find a measure which made a notable overlap with its curve. In contrast, an overlap occurs between the WongI and Tarantula curves, suggesting that if the user is to use either, the user should use Tarantula for the first 30% of the investigation, and if a bug has still not been found, continue with Wong-I. Observations of this sort provide helpful insights for the potential development of combination style algorithms like those of Kim [140].

Note that we included results in the performance plot for some measures based on their prominence and their ability to reveal some interesting trends, and in order to avoid general clutter. In general, many measures appear completely aligned (other than at microscopic examination) and thus it would not visually present any interesting results for the reader. We also include the random measure Rand as a baseline for comparison. Thus, for the Rand measure, after investigating 50% of the program you'd expect to have found a bug over 80% of the time on the benchmarks. The results of Rand are what we would expect with an average of 6 covered bugs per program.

We now discuss how the introduction of more faults affects single fault localisation effectiveness. To discuss this, we appeal to Figures 5.4 5.5 5.2 5.3, which represent how a range of measures perform as more bugs are introduced into the program. Here, the measures represented are chosen by their ability to illustrate general trends of all our measures.

In general, the more faults there were in a program the better an SBFL measure's W-scores, but the worse that measure's A-scores. A proposed explanation for this is that the more faults there were in a program, the more likely it was to find a fault early (due to increased luck – thus improving W-scores), but the less likely to find a fault immediately (due to increased noise – thus worsening A-scores). This confirms the results of Guiseppe et al [63]. An exception to this rule is for the single bug optimised measures (an example of which is given the Figures 5.4 5.5 5.2 5.3). Explanation for this is discussed below.

To ascertain the relationship between the number of faults in a program and the W-scores of $f9185$ (described in Figures 5.4 5.5 5.2 5.3), we used Spearman's rho. Here, the relationship between decreasing W-scores and increasing number of bugs was described as significant (R-score $<$ -0.99) for both the 4 SIR scores, and 6 Steimann scores. In addition, the relationship between decreasing A-scores and increasing number of bugs was described as significant (R-score $<$ -0.99) for only the Steimann scores, but not the SIR scores.

We now discuss how the size of the program affects fault localisation effectiveness. In general, in our top ten table there was a close relationship between performing well (in terms of rank) on the SIR benchmarks and performing well on the Steimann benchmarks. In particular, $m9185$ was first and second respectively for both SIR and Steimann benchmarks at overall W-score.

However, we noticed that the performance of a given measure appeared a lot better in the larger benchmarks than in the smaller in terms of the values presented by the AVG W and A scores. To see this, compare W and A scores in the top ten table, and compare 5.4 with 5.5, and 5.2 with 5.3). For instance, m9185 received a much better W score in the large benchmarks (2.63) than the small benchmarks (8.40). We believe this general trend is explained by two factors:

First, in theory if a bug is found on the nth investigation then this has a proportionally more dramatic effect on W-scores for small as opposed to large benchmarks. To see this, notice that finding a bug at the second point in a suspiciousness list for a program with 10 components gets W-scores of 10.00%. In contrast, these scores would be 1.00% in a program with 100 components. This means that measures would score a lot worse on smaller benchmarks even if they ranked a fault as the nth most suspicious component in a suspiciousness list. Accordingly, W scores can appear worse for smaller programs than larger.

Second, we observed the following property of our test suites: Many of the Steimann benchmarks had a lot of "short" failing traces (meaning they covered only a small proportion of components), which had the potential to result in a select few components being covered by more failing traces overall, therein often making measures weigh them as more suspicious to the advantage of fault localisation. In contrast, many benchmarks in the SIR suite (such as `tcas`) were such that each failing trace would cover most of the program, and there was a high tendency for many different blocks of code to have the same number of failing traces covering them, which in turn potentially worsened A scores. In summary, the test suites in the larger benchmarks were better for the purposes of fault localisation.

We now discuss the behaviour of single bug optimal measures. Our single bug optimised measures include the measures of the class $Opt(g)$ for some choice of suspiciousness measure g. The Naish measure is also single bug optimal. We acheived our best W score results for the $Opt(g)$ class when $g =$ Ochiai, which confirms our results of [149, 182]. In general, Table 5.9 shows that on AVG there is thus a tendency for single bug optimised measures to provide an advantage on our benchmarks in terms

of W scores, but to provide a disadvantage in terms of A scores. We discuss these observations as follows:

First, why do single bug optimised measures tend to have good W scores on AVG? Many bugs in our benchmarks, and presumably in practice, contain bugs which are covered by all failing traces (thus advantaging single bug optimal measures), and in the cases where this is not the case the number of program components which cover all failing traces are quite small (meaning that when a single bug optimal measure is disadvantaged, the disadvantage is often minimised). Despite the advantage, we observe that being single bug optimal is neither a necessary condition for high W scores (many of the other top performers are not single bug optimal but have quite high AVG W scores), nor is it sufficient (Naish is single bug optimal but has quite low AVG W scores).

Second, why do single bug optimised measures tend to have poor A scores on AVG? Table 5.10 demonstrates that single bug optimal measures are amongst some of the lowest scorers (the bottom half). We believe this is because when there are many bugs in the program, this raises the probability that the bugs are not covered by all failing traces, which in turn raises the chance that a single bug optimal measure does not investigate it first, even if it investigates it soon – leading to low A scores. In general, our comparative results for single bug optimised measures and their unoptimised counterparts reveal that if there is more than 1 bug in the program one is better using the unoptimised version as far as A scores are concerned (compare 5.4 with 5.5). We present the comparison between Opt-Ochiai and Ochiai in our figures as an example to demonstrate this point. In general, this provides us with the lesson that single bug optimised measures can come at a cost of performing less well when there are many bugs in the program, and therefore should be used be caution.

We now discuss some formal connections between top performing suspiciousness measures. We observe there is a formal connection between Kulkzynski2, Ochiai, and m9180, m30180. Following our ranking equivalence proofs in Section 5.2, we first observe that Ochiai and K2 are ranking equivalent to $P(C|E)P(E|C)$ and $P(C|E)+P(E|C)$ respectively. These are monotonically equivalent (divide each of the former expressions by 2) to the geometric and arithmetic mean of $P(C|E)$ and $P(E|C)$ respectively. We consequently say that $P(C|E)P(E|C)$ and $P(C|E) + P(E|C)$ describe *means of co-prediction*.

Furthermore, the top two performing automatically generated measures $m9185$ and $m30180$ contain $P(C|E) + P(E|C)$ and $P(C|E)P(E|C)$ as sub-expressions re-

spectively. Observe, $m9185 = (P(C|E) + P(E|C))$ - $\frac{P(\neg E|C)}{P(C|E)}$ and $m30180 = (P(E|C)P(C|E) + P(E|C)^2) + P(C|E)$.

Thus, we conclude that measures that are equivalent to a mean of co-prediction (or include such means as sub-expressions), are consistently top performing in terms of W-scores.

We now discuss additional experiments. An additional (smaller scale) experiment was performed with the Barinel tool, which was scalable for our SIR benchmarks but not our larger Steimann benchmarks, (see [21]). We performed this additional experiment because we were interested in whether sbfl techniques could compare to slightly more heavyweight fault localisation tools. The most recent version of the Barinel tool has an option to perform single fault localisation, but ranked 43rd overall in the SIR benchmarks in terms of W scores and was not competitive in this context. As a consequence we have not performed more detailed analysis of its performance. However, we emphasise that Barinel was designed originally as a multiple fault localisation tool (not a single bug localisation tool), and so shall compare Barinel in detail in our chapter on multiple fault localisation (chapter 7).

Finally, we discus the issue of how to determine the "best" measures. Although $m9185$ experimentally performs the best in our experiments in terms of AVG A and W scores, it is subject to criticisms of both over-fitting and not having a wholly satisfactory intuitive connection to the problem. Accordingly, criteria for finding "best" new measures might potentially include achieving a good balance of both experimental performance and a potentially more intuitive connection to sbfl. How one determines what this "best balance" is is a difficult question. But for brevity, we suggest that these measures might include the following classes of measures which were demonstrated to be experimentally competitive (top 25 W scorers as presented in Table 5.9):

1. New simple measures such as Pattern-similarity and NPV – demonstrated via our ranking equivalence proofs to be equivalent to $-a_{ep}a_{nf}$ and a_{np}/a_{nf} respectively.

2. New causal measures such as Suppes $= P(E|C) - P(E|\neg C)$ and Lewis $= \log \frac{P(E|C)}{P(E|\neg C)}$, which capture the intuition that "bugs raise the probability" of errors.

3. Established measures such as Kulkzynski2 and Ochiai, which capture the intuition that bugs and errors "co-predict" – demonstrated via our ranking equiva-

lence proofs to be equivalent to $P(E|C) + P(E|C)$ and $P(E|C)P(E|C)$ respectively.

5.4 Summary

In this chapter we have performed a large scale investigation into the established framework of SBFL. The major research project was to find the most effective suspiciousness measures. To answer this problem, we firstly introduced and motivated a large class of measures to SBFL – including classes of causal, confirmation, prediction, similarity, and automatically generated measures, and provided a large class of ranking equivalence proofs to identify new usable measures. We motivated these measures in terms of different proposals about what an SBFL suspiciousness measure should be designed to capture. Secondly, to find the most effective measures we provided what is to our knowledge the largest experimental comparison to date. We found our new automatically generated measure $m9185$ achieved the best AVG A and W-scores, and that many other of our new measures (e.g. Lewis, Pattsim, Suppes, and NPV) were competitive. Insights included that simple measures, measures of "co-prediction", and measures which capture the intuition that "bugs raise the probability" of errors perform well experimentally and have an improved intuitive connection to the problem.

In general, the large scale nature of our investigation may suggest that we have approached saturation point for new measures for the SBFL framework, and that if gains are to be made in terms of effectiveness, a major paradigm shift is required. Thus the question is begged: can we design a new framework which maintains the low cost running time of SBFL, but is superior - both in terms of exploiting newly identified theoretically properties of faulty programs, and in terms of practical fault localisation effectiveness? In the next chapter we address this question.

Chapter 6

Probabilistic Fault Localisation

In this chapter we advance the state-of-the-art in lightweight statistical fault localisation by building a new framework which we call probabilistic fault localisation (PFL). The framework is designed to overcome a potential theoretical shortcoming of SBFL. That is, so far there have not been many desirable formal properties which SBFL measures have been shown to satisfy. Although there has been some discussion of a few properties that measures should satisfy a priori (such as strict rationality [179]), measures that solve fault localisation sub-problems have been discussed (such as single bug optimal measures [181]), and general themes have been presented (such as bugs being "probability raisers", or "co-predictors", as per the previous chapter), there is not yet a SBFL measure that "solves" the problem of fault-localisation for all benchmarks in a meaningful sense, or has been demonstrated to experimentally outperform other measures by a substantial margin for given fault-localisation tasks. Indeed, recently Yoo et al. have established theoretical results which show that a "best" performing suspicious measure does not exist [253].

A proposed solution to this problem is to develop a new, comparably efficient alternative to SBFL which can at once exploit further properties key to fault localisation and experimentally outperform it at fault localisation tasks. Following this proposal, our general strategy is to develop a framework which can directly determine the probability that a given program artefact is faulty using the methods of probability theory. The contributions of this chapter are as follows:

- We introduce and motivate a new formal framework denoted *Probabilistic Fault Localisation* (PFL), which is designed to estimate the probability that a given program component is faulty. The framework is similar to SBFL insofar as it can leverage any given suspiciousness measure.

- We formally prove that PFL satisfies desirable formal properties which SBFL does not.
- We demonstrate that PFL techniques are substantially and statistically significantly more effective (using $p = 0.01$) than all known SBFL measures at finding faults on our large benchmarks. The user investigates 37% more code (and finds a fault immediately in 27% fewer cases) when using the best performing SBFL measures in the current literature.
- We show that it is theoretically impossible to define strictly rational SBFL measures that can outperform given PFL techniques on many of our benchmarks.
- We demonstrate that PFL maintains efficiency comparable to SBFL, and in experimentation is shown to provide fault localisation information in under a couple of seconds on average using our implementation.

The rest of this chapter is organised as follows. In Section 6.1 we present a motivating example. In Section 6.2, we introduce and motivate formal assumptions underlying the approach. In Section 6.3 we show that a set of equations, called the PFL-equations, follow from the assumptions of the previous section. In Section 6.4 we motivate a measure to be used in our framework. In Section 6.5, we formally prove that the PFL equations satisfy newly identified desirable properties, and also prove that the SBFL framework cannot satisfy these properties. In Section 6.6 we introduce and motivate a new algorithm, called the PFL-algorithm, which implements our PFL-equations for the practical task of fault localisation. Section 6.7 presents our experimental results of PFL techniques against all known SBFL measures at the task of finding a fault in a program using our benchmarks. In Section 6.8 we summarise our findings. This chapter is based on the paper of Landsberg et al. [148].

6.1 Motivating Example

We briefly re-present a small example to illustrate SBFL in the context of motivating the need for greater formal connection to the problem of fault localisation. Let us re-consider the faulty C program **minmax.c** in Figure 6.1 (from [103]). The faulty program is formally modelled as the following tuple of program components $\boldsymbol{PM} = \langle C_1, C_2, C_3, C_4, E \rangle$, where E models the event in which the specification `assert(least`

```
int main() {
  int input1, input2, input3;
  int least = input1;
  int most = input1;

  if (most < input2)
    most = input2;  // C1
  if (most < input3)
    most = input3; // C2
  if (least > input2)
    most = input2; // C3 (fault)
  if (least > input3)
    least = input3; // C4

  assert(least <= most); // E
  }
```

Figure 6.1: `minmax.c`

	C_1	C_2	C_3	C_4	E
t_1	0	1	1	0	1
t_2	0	0	1	1	1
t_3	0	0	1	0	1
t_4	1	0	0	0	0
t_5	0	1	0	0	0
t_6	0	0	0	1	0
t_7	0	0	1	1	0
t_8	0	0	0	0	0
t_9	1	0	0	1	0
t_{10}	1	1	0	0	0

Figure 6.2: coverage matrix

`<= most)` is violated. The program fails to always satisfy this specification. The explanation for the failure is the fault at C_3, which should be an assignment to `least` instead of `most`. We collected coverage data from ten test cases to form our set of coverage vectors $\mathbf{T} = \{t_1, \ldots, t_{10}\}$. The coverage matrix for these test cases is given in Table 6.2. Three of the test cases fail and seven pass. We compute the program spectrum for each component using the coverage matrix. For example, the program spectrum for C_3 is $\langle 3, 0, 1, 6 \rangle$.

To illustrate an instance of SBFL we use the suspiciousness measure Wong-2 $= a^i_{ef} - a^i_{ep}$ [241]. The user inspects the program in decreasing order of suspiciousness until a fault is found. C_3 is inspected first with a suspiciousness of 2 and thereby a fault is found immediately. The example illustrates that SBFL measures can be successfully employed as heuristics for fault localisation, but that the formal and conceptual connection to fault localisation could potentially be improved.

6.2 Assumptions

Our goal is to define a probability function which describes the probability of the hypothesis that a given component is faulty as a function of a given proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$. This information can then be used in practical fault localisation tasks by suggesting to the user which parts of the faulty program to investigate first in

the search for faults. In the remainder of this section we do the following. We first describe an intuition for the sort of probability function we wish to define, we then present some formal preliminaries, and finally we introduce, motivate, and formally present some assumptions. These assumptions are then are used to derive some fault localisation equations of section 6.3, which are in turn used to determine the probability that a given component is faulty.

We first discuss how we intend the probabilities of the proposed probability function to be interpreted. For our purposes, we wish the probabilities to describe the degree to which the empirical evidence supports the given hypothesis (in our case a given proband model will supply the empirical evidence, and the given hypothesis will be a proposition expressing whether a given UUT is faulty). This has been called the *logical*, *inductive*, or *epistemic* probability [86]. To gain an intuition for this type of probability, suppose I flip a coin, and cover the result of the flip with my hand, such the result of the flip is unknown to you. Now, objectively speaking the coin is either facing up heads, or facing up tails (but not both), and so in one sense the probability that the coin is facing up heads is either 1 or 0. However, according to the epistemic interpretation, the probability it is facing heads up describes the degree to which the empirical evidence supports that hypothesis. Now, as the degree to which the evidence supports that hypothesis is equal to the degree to which the evidence supports the contrary hypothesis that the coin is facing up tails, and that either (but not both) of these hypotheses must be true, the epistemic probabilities would describe this by assigning an equal probability to both hypotheses (0.5 to each).

To see how this intuition might apply to our development, consider a proband which consists of a faulty program (consisting of two UUTS) and a test suite which consists of only one failing execution which covers the entire program. For this proband, we may construct the proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ consisting of the program model $\boldsymbol{PM} = \langle C_1, C_2, E \rangle$ and set of coverage vectors $\mathbf{T} = \{t_1\}$ in which $t_k = \langle 1, 1, 1, k \rangle$ where $k = 1$. Now, assuming that either C_1 or C_2 (but not both) indicate the cause of the error, we can estimate the likelihood that one indicates the cause using our proband model as our available evidence. In this example, we think the degree to which the evidence supports the claim that C_1 indicates the cause, is equal to the degree to which the evidence supports the claim that C_2 indicates the cause (as the two are indistinguishable in the program model apart from their ordering – where we assume that the ordering does not provide any evidence here). According to an epistemic interpretation of probability the probabilities would describe this by assigning an equal probability to both hypotheses.

We now present some formal preliminaries. Alongside the probability function defined in section 3.4 of chapter 3, we introduce a new probability function P_2 the domain of which is a set of propositions. To define the set of propositions, we first define two sets of atomic propositions $\mathbf{H} = \{h_i | C_i \in \boldsymbol{PM}\}$ and $\mathbf{C} = \{h_i^k | C_i \in \boldsymbol{PM} \wedge t_k \in \mathbf{T}\}$. Intuitively, $\mathbf{H}$ is a set of fault hypotheses, where h_i expresses the hypothesis that C_i is faulty, and $\mathbf{C}$ is a set of causal hypotheses, where h_i^k expresses the hypothesis that C_i was the cause of the error E in execution t_k. For the purposes of our development, members of $\mathbf{H} \cup \mathbf{C}$ are treated as definitionally primitive propositions and are not defined in terms of any further formal object. The set of propositions is then defined inductively as follows. For each $p, q \in \mathbf{H} \cup \mathbf{C}$, p and q are propositions. If p and q are propositions, then $p \wedge q$, $p \vee q$, $\neg p$ are propositions. We also assume the following standard properties of probability [117]. For each proposition p and q: $P_2(p) = 1$ if $p = \top$. $P_2(p) = 0$ if $p = \bot$. $P_2(p \vee q) = P_2(p) + P_2(q) - P_2(p \wedge q)$. $P_2(\neg p) = 1 - P_2(p)$. $P_2(p|q) = P_2(p \wedge q)/P_2(q)$.

We now present assumptions **A1-7** which are designed to be plausible for any proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$. These assumptions will be used to generate a set of three equations which can be used to achieve to determine the probability that any given component is faulty.

A 1. For all $h_i \in \mathbf{H}$, $h_i = \bigvee_{k=1}^{|\mathbf{T}|} h_i^k$.

This states that C_i is faulty just in case C_i was the cause of the error E in some execution of the program, and provides a fundamental definition of what it is to be faulty. An assumption underlying **A1** is that the given test suite is adequate in the following sense: that for every fault there is a test case in which that fault is executed and can be identified as the cause of the error.

A 2. For all $t_k \in \mathbf{F}$, $\bigvee_{i=1}^{|\boldsymbol{PM}|} h_i^k = \top$.

This states that for every failing trace, there is some component $C_i \in \boldsymbol{PM}$ which caused the error E in that trace. Intuitively, we assume that if an error occurred then something (some event in the program) must have caused it, and that no error occurs in a failing trace which can be said to be uncaused. In general we also assume that $\boldsymbol{PM}$ is defined adequately in the sense that every fault has a corresponding component in $\boldsymbol{PM}$.

A 3. For all $h_i^k \in \mathbf{C}$, if $h_i^k = \top$ then $C_i \neq E$.

A 4. For all $h_i^k \in \mathbf{C}$, if $h_i^k = \top$ then $c_i^k = 1$ and $e^k = 1$.

These assumptions state that if C_i was the cause of E in t_k, then C_i must have been a different event to E (**A3**), and C_i and E must have actually occurred (**A4**). These two assumptions have been described as fundamental properties about causation [155].

A 5. For all $h_i^k, h_j^k \in \mathbf{C}$, if $C_i \neq C_j$ then $h_i^k \wedge h_j^k = \bot$.

This states that no two events could have both been the cause of the error in a given trace. In other words, different causal hypotheses for the same trace are mutually exclusive. The rationale for this is that the intended and fundamental meaning of h_i^k is C_i was *the* cause of E in t_k, and as *the* implies uniqueness, no two events could have been *the* cause. In general, any union of events may be said to be *the* cause so long as that union is in $\boldsymbol{PM}$.

A 6. For all $h_i^k \in \boldsymbol{C}$ and all $\mathbf{S} \subseteq \mathbf{T} - \{t_k\}$, $P_2(h_i^k | \bigvee_{t_n \in S} h_i^n) = P_2(h_i^k)$.

This states that the probability that C_i was the cause in one trace is not affected by whether it was in some others. In other words, whether it was the cause in one is statistically independent of whether it was in others. Here, we assume that our probabilities describe this property of objective chance, and that the causal properties of each execution is determined by the properties of the events in that execution alone, cannot affect the causal properties of other executions, and so do not affect the probabilities of those causal properties. Independence principles are well established in probability theory [139].

In light of the above assumptions we may define $c(t_k) = \{C_i \in \boldsymbol{PM} | c_i^k = e^k = 1 \wedge C_i \neq E\}$. Intuitively, this is the set of components covered in t_k, otherwise known as the set of *candidate causes* of E in t_k. We now state the following assumption:

A 7. For some measure w and for all $C_i, C_j \in c(t_k)$, $P_2(h_i^k)/P_2(h_j^k) = w(C_i)/w(C_j)$.

Here, we assume w is a correct measure of the strength/propensity of a given event to cause the error (and is thus motivated as a measure of causal strength as described in section 5.1.3). We also assume there exists a correct measure and that this measure can be found. Accordingly, the assumption states that there is some measure of causal propensity such that the relative likelihood that one event caused the error over another, is directly proportional to their causal propensities to do so. In general, any suspiciousness measure w from the SBFL literature may be proposed as a measure of causal strength, and thus there is great room for experimentation over the definition of w. We use the notation PFL-w when measure w is being used.

Two provisos on the definition of w come from whether C_i is known to be faulty or not. Firstly $w(C_i) > 0$ if $a^i_{ef} > 0$ when C_i is not known to be non-faulty (this proviso may require some measures to be re-scaled). This corresponds to the intuition that there is some chance that C_i is faulty when C_i is not already known to be healthy and is involved in some failing traces. It also helps ensure there are no divisions by zero. Secondly, $w(C_i) = 0$ if C_i is already known to be non-faulty. This corresponds to the intuition that C_i has no propensity, and consequently no probability, in being the cause of the error if this has already determined by the user to be the case. We propose a definition for w in section 6.4.

6.3 PFL-Equations

We now show that the assumptions **A1-7** (henceforth PFL assumptions) imply Equations (6.1), (6.2), and (6.3) (henceforth PFL equations. The PFL equations can be used to determine the probability that a given component C_i is faulty by finding the value of $P_2(h_i)$. We may informally describe these three equations as follows. The first equation states that the probability a component is faulty, is the probability that it was the cause of the error in some trace. The second is a general definition of disjunction under the assumption of independence. The third states that the probability that a covered component was the cause of the error in a given trace is equal to its normalised suspiciousness (or 0 if the component is not covered in that trace).

$$(\forall h_i \in \mathbf{H}) P_2(h_i) = P_2(\bigvee_{n=1}^{|\mathbf{T}|} h_i^n) \tag{6.1}$$

$$(\forall h_i^k \in \mathbf{C}) P_2(\bigvee_{j=k}^{|\mathbf{T}|} h_i^j) = P_2(h_i^k) + P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_i^j) - P_2(h_i^k)\, P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_i^j) \tag{6.2}$$

$$(\forall h_i^k \in \mathbf{C}) P_2(h_i^k) = \begin{cases} \frac{w(C_i)}{\sum_{C_j \in c(t_k)} w(C_j)} & \text{if } C_i \in c(t_k) \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

Proposition 2. The PFL assumptions imply the PFL equations.

Proof. We first show Equation (6.1). It suffices to prove the case for h_1. $h_1 = \bigvee_{k=1}^{|\mathbf{T}|} h_1^k$ (by **A1**). Thus $P_2(h_1) = P_2(\bigvee_{k=1}^{|\mathbf{T}|} h_1^k)$ (by Leibniz's law).

We now show Equation (6.2). It suffices to prove the case for each $h_1^k \in \mathbf{C}$. By the definition of disjunction we have for all $0 < k \leqslant |\mathbf{T}|$ $P_2(\bigvee_{j=k}^{|\mathbf{T}|} h_1^j) = P_2(h_1^k) + P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j) - P_2(h_1^n \wedge \bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j)$. It remains to show $P_2(h_1^k \wedge \bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j) = P_2(h_1^k) P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j)$. $P_2(h_1^k \wedge \bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j)$ is equal to $P_2(h_1^k | \bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j) P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j)$ (by probabilistic calculus). This is equal to $P_2(h_1^k) P_2(\bigvee_{j=k+1}^{|\mathbf{T}|} h_1^j)$ (by **A6**).

We now show Equation (6.3). We have two cases to consider: $C_i \in c(t_k)$ and $C_i \notin c(t_k)$. Assume $C_i \in c(t_k)$. We may assume t_k is ordered so there is some n such that $\bigwedge_{i=1}^{n} c_i^k = 1$, $\bigwedge_{i=n+1}^{|\boldsymbol{PM}|-1} c_i^k = 0$ and $c_{|\boldsymbol{PM}|}^k = e^k = 1$ (such that $c(t_k) = \{C_1, \ldots, C_n\}$). Now, for all $C_i, C_j \in c(t_k)$ $P_2(h_i^k)/P_2(h_j^k) = w(C_i)/w(C_j)$ (by **A7**). Thus, for $C_i, C_j \in c(t_k)$ $w(C_i)/P_2(h_i^k) = w(C_j)/P_2(h_j^k)$ (as $x/y = w/z \equiv z/y = w/x$). So, $w(C_1)/P_2(h_1^k) = w(C_2)/P_2(h_2^k) = \ldots = w(C_n)/P_2(h_n^k)$. Thus, there is some c such that for all $C_i \in c(t_k)$, $c = w(C_i)/P_2(h_i^k)$ (by the last result). Equivalently, there is some c such that for all $C_i \in c(t_k)$, $P_2(h_i^k) = w(C_i)/c$. To complete the proof it remains to prove $c = \sum_{C_j \in c(t_k)} w(C_j)$. $\bigvee_{i=1}^{|\boldsymbol{PM}|} h_i^k = \top$ (by **A2**). But, $\bigvee_{i=n+1}^{|\boldsymbol{PM}|-1} h_i^k = \bot$ (by **A4** and contra-position with the initial assumption that $\bigwedge_{i=n+1}^{|\boldsymbol{PM}|-1} c_i^k = 0$), and $h_{|\boldsymbol{PM}|}^k = \bot$ (by **A3**). Thus, $\bigvee_{i=1}^{n} h_i^k = \top$ (by $\vee$-elimination). So, $P_2(\bigvee_{i=1}^{n} h_i^k) = 1$ (by probabilistic calculus). Thus, $\sum_{i=1}^{n} P_2(h_i^k) = 1$ (by probabilistic calculus and **A5**). So, $\sum_{i=1}^{n}(w(C_i)/c) = 1$. Thus, $(\sum_{i=1}^{n} w(C_i))/c = 1$. Equivalently, $\sum_{i=1}^{n} w(C_i) = c$. So, $\sum_{C_i \in c(t_k)} w(C_i) = c$ (by def. of $c(t_k)$ above). We now do the second condition. Assume $C_i \notin c(t_k)$. Then $\neg(c_i^k = e^k = \top \wedge C_i \neq E)$ (by def. of $c(t_k)$). Thus $c_i^k = 0$ or $e^k = 0$ or $C_i = E$. If $C_i \neq E$, then $P_2(h_i^k) = 0$ (by **A3**). If $c_i^k = 0$, then $P_2(h_i^k) = 0$ (by **A4**). If $e^k = 0$, then $P_2(h_i^k) = 0$ (by **A4**). Thus, if $C_i \notin c(t_k)$, then $P_2(h_i^k) = 0$.

□

Example 6.3.1. We now illustrate how the PFL equations can be used to determine the value $P_2(h_4)$ from our running example of 3.1. We use a weight $w = 1$ for simplicity. In this example, $P_2(h_4)$ is equal to $P_2(h_4^1 \vee h_4^2 \vee h_4^3)$ (by Equation (6.1)). It is sufficient to first find the values of each of $P_2(h_4^1)$, $P_2(h_4^2)$ and $P_2(h_4^3)$, and proceed from there. We do the case for $P_2(h_4^2)$ explicitly. This is equal to $w(C_4)/(w(C_3)+w(C_4))$ (by Equation (6.3)) which is equal to $\frac{1}{1+1}$ (as w returns 1 in our example). Thus, $P_2(h_4^2) = 0.5$. Finally, $P_2(h_4^1) = 0$ and $P_2(h_4^3) = 0$ because C_4 is neither covered in t_1 nor t_3. Now, $P_2(h_4^2 \vee h_4^3)$ is equal to $P_2(h_4^2) + P_2(h_4^3) - P_2(h_4^2)P(h_4^3)$ (by Equation (6.2)). This is equal to $1/2 + 0 - ((1/2)0) = 1/2$ (by substitution). Furthermore, $P_2(h_4^1 \vee h_4^2 \vee h_4^3)$ is equal to $P_2(h_4^1) + P_2(h_4^2 \vee h_4^3) - P_2(h_4^1)P_2(h_4^2 \vee h_4^3)$ (by Equation (6.2)). This is equal to $0 + 1/2 - (0(1/2)) = 1/2$. Thus $P_2(h_4^1 \vee h_4^2 \vee h_4^3)$ is equal to 0.5 $P_2(h_4^1 \vee h_4^2 \vee h_4^3)$ is equal to $P_2(h_4)$ (by Equation (6.1)). Thus, $P_2(h_4) = 0.5$. By similar reasoning we have $P_2(h_2) = 0.5$ and $P_2(h_3) = 1$. In contrast, PFL-PPV returns $P_2(h_2)$, $P_2(h_3)$, $P_2(h_4)$

values of 0.31, 1.00, 0.25 respectively. For a case study which demonstrates the use of the PFL-equations in the context of constructing a mass function for a program in our large benchmarks, the reader is directed to Appendix A.4.

6.4 Measure of Causal Propensity

To use the PFL equations, it remains for the user to choose a measure of causal propensity w for **A7**. One proposal is $w(C_i) = P_1(E|C_i)$ (the measure of positive predictive power PPV [149]) or $P_1(E|C_i)/P_1(E)$ (the Fitelson measure of causal strength [78]). For the purposes of defining $P_2(h_i^k)$ both proposals are equivalent (observe $P_2(h_i^k)/P_2(h_j^k) = P_1(E/C_i)/P_1(E/C_i) = (P_1(E/C_i)/P_1(E))/(P_1(E/C_j)/P_1(E))$ using **A7**).

The proposal satisfies the following plausible intuitions about causal likelihood. As follows:

1. It captures the intuition that the more something raises the probability of the error, the higher its relative likelihood that it is the error's cause (to see this, observe we have $P_2(h_i^k)/P_2(h_j^k) = P_1(E|C_i)/P_1(E|C_j)$ using **A7**).

2. It captures the intuition that events which do not affect the error's likelihood are equally unlikely to have caused it (to see this, assume both C_i and C_j are independent of E i.e. $P_1(E) = P_1(E|C_i)$ and $P_1(E) = P_1(E|C_j)$, then it follows $P_2(h_i^k) = P_2(h_j^k)$ using **A7**).

3. A plausible estimate of $w(C_i)$ as a measure of C_i's causal strength is the probability that C_i causes E given C_i, and $P_1(E|C_i)$ accordingly provides an upper bound for this estimate.

4. The proposal has the following intuitive consequence: when there are no passing traces, then for every C_i and t_k we have $P_2(h_i^k) = 1/n$ for n covered components in trace t_k. This corresponds to a uniform probability distribution for all covered components in each failing trace. To see this result, assume $a_{ep}^i = 0$ for every $C_i \in \boldsymbol{PM}$, then PPV $= a_{ef}^i/(a_{ef}^i + a_{ep}^i) = 1$ for every C_i, in which case $P_2(h_i^k) = 1/n$ (by equation 3 of the PFL-equations).

In our running example PFL-PPV returns $P_2(h_2) = 0.00$, $P_2(h_2) = 0.31$, $P_2(h_3) = 1.00$, and $P_2(h_4) = 0.25$, which correctly identifies the correct hypothesis with the most probable one.

Finally, we emphasise that the correct definition of causal strength w is subject to debate (as established by the current literature [78]). Consequently, there is room for experimentation over alternative definitions.

6.5 Properties

We now identify desirable formal properties which we prove the PFL equations satisfies, but no SBFL suspiciousness measure can.

Definition 6.5.1. *Fault Likelihood Properties.* For all $C_i, C_j \in \boldsymbol{PM}$, where $C_i \neq C_j$, we define the following:

1. Base case. If there is some failing trace which only covers C_i, but this property does not hold of C_j, then C_i is more suspicious than C_j
2. Extended case. Let $\mathbf{T}_1$ be a test suite in which all failing traces cover more than one component, and let $\mathbf{T}_2$ be identical to $\mathbf{T}_1$ except $c_i^k = 1$ and $c_j^k = 1$ in $\mathbf{T}_1$ and $c_i^k = 1$ and $c_j^k = 0$ in $\mathbf{T}_2$, then the suspiciousness of C_i in $\mathbf{T}_2$ is more than its suspiciousness in $\mathbf{T}_2$.

These properties capture the intuition that the fewer covered components there are in a failing trace, the fewer places there are for the fault to "hide", and so the a priori likelihood that a given covered component is faulty must increase. Upper bounds for this increase is established by the base case – if a failing trace only covers a single component then that component must be faulty. We now formally establish that the PFL equations, but no SBFL measure, satisfies these properties.

Proposition 3. The PFL equations satisfies the fault likelihood properties.

Proof. We first prove the base property. We first show that if there is some failing trace t_k which only covers C_i, then nothing is more suspicious than it. Let t_1 be a failing trace which only covers C_i. Then $P_2(h_i^1) = \frac{w(C_i)}{w(C_i)} = 1$ (by Eq. (6.3)). Letting n abbreviate $P_2(\bigvee_{j=2}^{|\mathbf{T}|} h_i^j)$, we then have $P_2(\bigvee_{k=1}^{|\mathbf{T}|} h_i^k) = (1+n)-(1n) = 1$ (by Eq. (6.2)). So $P_2(h_i) = 1$ (by Eq. (6.1)). Thus, nothing can be more suspicious than C_i. We now show that if there is no failing trace which only covers C_j, then C_j must be less suspicious than C_i. Assume the antecedent, then for each t_k we have $P_2(h_i^k) = \frac{w(C_i)}{w(C_i)+\cdots+w(C_k)} < 1$ (by Eq. (6.3)). Thus $P_2(\bigvee_{k=1}^{|\mathbf{T}|} h_i^k) < 1$ (by Eqs. (6.2) and (6.3)).

Thus $P_2(h_j) < 1$ (by Eq. (6.1)). Thus $P_2(h_j) < P_2(h_i)$, which means C_i is more suspicious than C_j.

We now prove the extended property. Let $\mathbf{T}_1$ be a test suite in which all failing traces cover more than one component, and let $\mathbf{T}_2$ be identical to $\mathbf{T}_1$ except $c_i^1 = 1$ and $c_j^1 = 1$ in $\mathbf{T}_1$ and $c_i^1 = 1$ and $c_j^1 = 0$ in $\mathbf{T}_2$. Let n abbreviate $P_2(\bigvee_{m=2}^{|\mathbf{T}|} h_i^m)$ $P_2(h_i) = P_2(h_i^1) + n - (P_2(h_i^1)n)$ (by Eqs. (6.1) and (6.2)). It remains to first show that $P_2(h_i^1)$ is greater in $\mathbf{T}_2$, and secondly show n has the same value for both test suites where $n < 1$. For the former, let $P_2(h_i^1) = \frac{w(C_i)}{w(C_1)+\cdots+x+\cdots+w(C_{|c(t_k)|})}$ for both test suites (using Eq. (6.3)), where we let $x = w(C_j)$ for $\mathbf{T}_1$ (where $w(C_j) > 0$), and $x = 0$ for $\mathbf{T}_2$ (as $c_j^k \notin c(t_k)$ for $\mathbf{T}_2$). So, the equation for $P_2(h_i^1)$ is greater in $\mathbf{T}_2$. To show the latter, we observe that for all $1 < m \leq |\mathbf{T}_1|$ we have $P_2(h_i^m) < 1$ (by assumption each $t_m \in \mathbf{F} \subseteq \mathbf{T}_1$ covers at least 2 components) and that $P_2(h_i^m)$ is the same in both $\mathbf{T}_1, \mathbf{T}_2$, thus $n < 1$ (by Eq. (6.2)) and n has the same value for both. □

Proposition 4. No SBFL measure satisfies either property.

Proof. To show that no suspiciousness measure w satisfies the base property, we show that for any w we can construct a test suite in which 1) there is a failing trace which only covers C_i, 2) there is some C_j such that there is no failing trace which only covers it, and 3) $w(C_i) = w(C_j)$. A simple example is as follows. Let $\boldsymbol{PM} = \langle C_1, C_2, C_3, E\rangle$ and $\mathbf{T} = \{\langle 1,1,1,1,1\rangle, \langle 0,1,1,1,2\rangle, \langle 1,0,0,1,3\rangle\}$. Thus the spectrum for C_1 and C_2 is $\langle 2,0,0,0\rangle$, and so $w(C_1) = w(C_2)$.

To show that no suspiciousness measure w satisfies the extended property, we show that for any w we can construct a pair of test suites $\mathbf{T}_1$ and $\mathbf{T}_2$ which are otherwise identical except 1) $c_i^k = 1$ and $c_j^k = 1$ in $\mathbf{T}_1$ 2) $c_i^k = 1$ and $c_j^k = 0$ in $\mathbf{T}_2$, and 3) $w(C_i)$ is equal to $w(C_j)$. The simplest example is as follows. Let $\boldsymbol{PM} = \langle C_1, C_2, E\rangle$ and $\mathbf{T}_1 = \{\langle 1,1,1,1\rangle\}$ and $\mathbf{T}_2 = \{\langle 1,0,1,1\rangle\}$. Thus the spectrum for C_1 is $\langle 1,0,0,0\rangle$ in both cases, and thus $w(C_1)$ is the same in both cases.

□

The proof of the last proposition suggests that there are large classes of test suites in which SBFL measures violate the properties. SBFL measures do not have the resources to satisfy the properties because each C_i's suspiciousness is only a function of its program spectrum, which itself is only a function of the i-th column of a coverage matrix.

6.6 PFL-Algorithm

In this section we present an algorithm which uses the PFL-equations in the context of finding a fault in a faulty program. We call this new algorithm the PFL-algorithm.

We begin with some development. A first question is how to use the PFL equations in the context of a fault localisation algorithm. One first proposal is to use the traditional SBFL algorithm, and use the PFL-equations to determine the degree of suspiciousness for each component. After each component has its fault probability computed, the user then inspects the program in descending order of suspiciousness until the fault is found. The reason the SBFL should not be used is due to the fact that in our setup the probability that a given component is faulty can change when new information about the fault status of a component is discovered (namely, $w(C_i)$ is set to 0 if C_i is discovered to be non-faulty), therein changing the order in which components should be investigated (an example of this is given later).

Accordingly, we propose a new algorithm as follows. Given a proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ and measure w, an algorithm to find a single fault in a program $\boldsymbol{PM}$ is as follows. Step one, find $\max_{h_i \in \mathbf{H}}(P_2(h_i))$ by computing the value of $P(h_i)$ for each $h_i \in \mathbf{H}$ using the PFL equations. Let the most probable hypothesis be h_j. If h_j is true, the procedure stops. Otherwise set $w(C_j) = 0$ and return to step one. A property of this algorithm is that yet to be investigated components can change in fault likelihood at each iteration when new facts of the form $h_j = \bot$ are discovered. We formally present step one of this algorithm in the PFL-algorithm. Informally, this algorithm can be intuitively understood as computing the most likely fault hypothesis using the three fault localization equations presented in section 6.3 and a given proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$.

One optimisation in the algorithm is as follows. A consequence of the PFL equations is that we need only compute the values $P_2(h_i^k)$ for each failing trace t_k and component C_i. This is because $P_2(h_i^k) = 0$ for passing traces, and so passing traces have no impact on the computation of $P(h_i)$.

We now discuss complexity. It is easily observed that the complexity of the PFL algorithm can be described as a linear function of the size of the test suite $\mathbf{T}$. It is therefore comparably lightweight to SBFL.

We emphasise that a (potentially advantageous) property of the PFL-algorithm is that fault probabilities can *update* after each investigation. We illustrate this with two examples. The first shows how absolute values of probabilities can change; the

Algorithm 1 PFL-algorithm

Input: proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$, and measure w
Output: index of the most likely fault hypothesis

```
for 0 < i < |PM| do
    P[i] ← 0
end for
for 0 < k ≤ |F| do
    sum ← 0
    for 0 < i < |PM| do
        if c_i^k = 1 then
            sum ← sum + w(C_i)
        end if
    end for
    for 0 < i < |PM| do
        if c_i^k = 1 then
            P[i] ← (P[i] + (w(C_i)/sum)) − (P[i] ∗ (w(C_i)/sum))
        end if
    end for
end for
return argmax_i(P[i])
```

	C_1	C_2	C_3	C_4	E
t_1	1	1	0	0	1
t_2	1	0	1	1	1
t_3	1	0	1	1	1

Table 6.1: coverage matrix, small example

second shows how the relative order in which hypotheses are ranked in terms of suspiciousness can change.

First, we give an example of how updates can mean that hypotheses change in *absolute* value. To see this, consider a program $\boldsymbol{PM} = \langle C_1, C_2, C_3, C_4, E \rangle$ where $\mathbf{T} = \mathbf{F} = \{\langle 1, 1, 1, 1, 1, 1 \rangle\}$, and where C_2 is a bug. Using PFL-PPV we first observe that each program component has a probability 1/4. Suppose it is then discovered that C_1 is not a fault. Then, as $w(C_1)$ is now set to 0, the probability of each of the remaining hypotheses is 1/3. Thus, the probabilities have changed in absolute value.

Secondly, we give an example of how updates can mean that hypotheses change in *relative* order of suspiciousness after an investigation of the program. For instance, there are cases in which if C_3 is more likely than C_2, then after C_1 has been discovered to be not faulty then C_2 will be more likely than

C_3. To see this, consider a program $\boldsymbol{PM} = \langle C_1, C_2, C_3, C_4, E\rangle$ where $\mathbf{T} = \mathbf{F} = \{\langle 1,1,0,0,1,1\rangle, \langle 1,0,1,1,1,2\rangle, \langle 1,0,1,1,1,3\rangle\}$, and where C_2 is a bug, and one of C_3 or C_4 is a bug. We present this test suite in Table 6.1. In this example, the PFL algorithm runs as follows. At step one PFL-PPV gives us $P_2(h_1) > P_2(h_3)|P_2(h_4) > P_2(h_2)$. Thus C_1 is investigated first. As it is not faulty $w(C_1)$ is consequently set to 0. PFL-PPV then gives us $P_2(h_2) > P_2(h_3)$. Thus, C_2 and C_3 has changed in ordering since the first investigation as a result of the update, and the bug (on average) is found quicker. In contrast, in SBFL degrees of suspiciousness remain invariant throughout the fault localisation process. In general, we believe that an algorithm which allows updates concerning the fault status of components has the potential to improve fault localization.

6.7 Empirical Evaluation

In this section we discuss our experimental setup and our results. The aim of the experiment is to compare the performance of the PFL algorithm against SBFL measures at the practical task of finding a fault in programs.

We first discuss techniques compared. We compared the experimental results of our SBFL techniques against a selection of PFL techniques. In addition, as we now want to determine whether PFL can outperform any rational SBFL measure in general, we also include the unavoidable cost (UC) scores. We now discuss which PFL techniques we compared. Although PFL techniques have a comparable complexity to SBFL techniques (taking a negligible number of seconds per problem in our implementation), a problem was that with 60,000 program versions running the experiment with just four PFL techniques took considerable time. For this reason, we had to pick wisely which values of w to test with PFL. We chose four measures, as follows. Firstly, we chose $PPV = P_1(E|C)$ (equivalent to Fitelson-I measure) following our discussion in section 6.4, we additionally chose scaled and ranking equivalent versions of Ochiai $= P_1(E|C)P_1(C|E)$ and Kulkzynski2 $= P_1(E|C) + P_1(C|E)$ and Suppes measure $(P_1(E|C) - P_1(E|\neg C) + 1)/2$. These measures are scaled from 0 to 1, and were chosen because of their promising experimental performance in SBFL, and because of their potential applicability as measures of causal power.

We first present the top ten performing measures in section 6.7.1 along with overall averages of the techniques compared. We then present scores for a range of methods as

Technique	AVG W	SIR W	STI W	AVG A	SIR A	STI A
UC	2.65	4.55	1.38	69.25	58.45	76.45
m9185	**4.93**	**8.40**	2.63	47.39	**38.15**	53.55
PFL-SUPPES	5.12	9.66	2.09	53.67	25.73	72.30
K2	5.12	8.76	2.69	45.09	33.75	52.65
PFL-OCHIAI	5.39	10.40	2.05	54.90	28.43	72.55
PFL-K2	5.50	10.27	2.31	52.11	24.43	70.57
PFL-PPV	5.69	11.39	**1.88**	**57.38**	29.15	**76.20**
Zoltar	6.09	11.34	2.59	43.26	28.10	53.37
Opt-Ochiai	6.38	10.95	3.34	37.75	21.80	48.38
D3	6.38	10.87	3.39	40.62	27.52	49.35
Ochiai	6.58	11.99	2.97	43.34	29.65	52.47

Table 6.2: PFL – top 10 W scorers

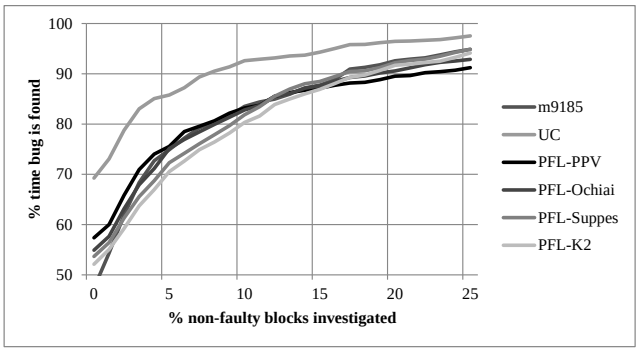

Figure 6.3: Performance of prominent PFL techniques

a function of the number of faults introduced into the program more in Section 5.3.2. Finally, we discuss these results in section 5.3.3.

6.7.1 Scores by average

To provide a summary of our top results, the measures which received the best overall AVG W-scores are presented in Table 6.2. The first column gives the name of the technique, the second/fifth columns give the AVG W/A scores respectively for all 100 scores. The remaining columns give the AVG W/A scores for the SIR and Stiemann (STI) benchmarks respectively. In Appendix A we provide detailed tables which describe the 100 scores for the n-fault benchmarks for a range measures for the interested reader.

We also present a performance plot in Figure 6.3.

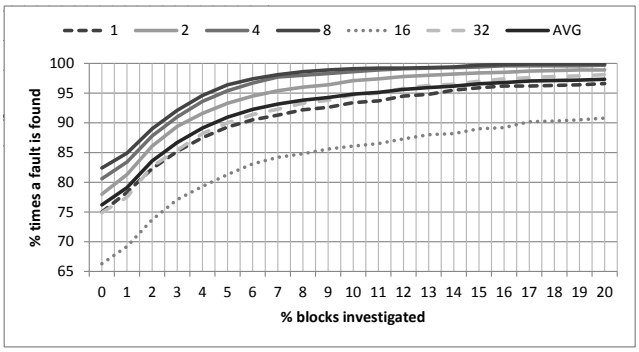

Figure 6.4: Performance of PFL-PPV on Steimann benchmarks

6.7.2 Scores by number of faults

In Figures 6.5, 6.6, 6.7, and 6.8 we present some detailed W-score and A-score results for a range of techniques. The figures are constructed analogously to those in section 5.3.2. In these figures we have chosen our favoured PFL technique PFL-PPV, and compared it against PPV when used as a SBFL measure (here we note that PPV is ranking equivalent to Tarantula, and so is reported as Tarantula in our graphs). We also chose the best performer in terms of W scores in the Stiemann benchmarks (Zoltar) and the best performer in terms of W scores in the SIR benchmarks (m9185). This selection firstly demonstrates how PFL-PPV improves in effectiveness of PPV, and secondly demonstrates how it often improves upon top performing SBFL measures both in theory (using a measure of the upper bounds for SBFL performance UC) and practice (using the best performing measures Zoltar and m9185).

6.7.3 Discussion

In this section we discuss the experimental results of the previous section. The central theme of the discussion concerns how PFL compares to SBFL framework in general, and in identifying the advantages the former has over the latter.

We begin by discussing how PFL compares to SBFL in terms of overall effectiveness. Table 6.2 demonstrates that our PFL techniques are extremely competitive with the SBFL framework overall. In general, all four of our tested PFL techniques are in the top 6 performers (as measured by overall AVG W score). In many cases PFL is demonstrated to be the top performer: The top overall AVG A score of any technique

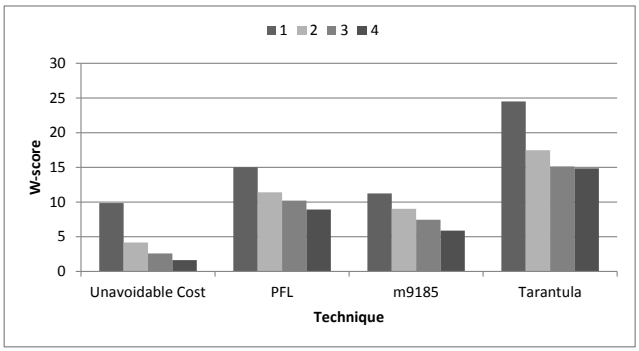

Figure 6.5: W-scores for selected techniques on SIR benchmarks

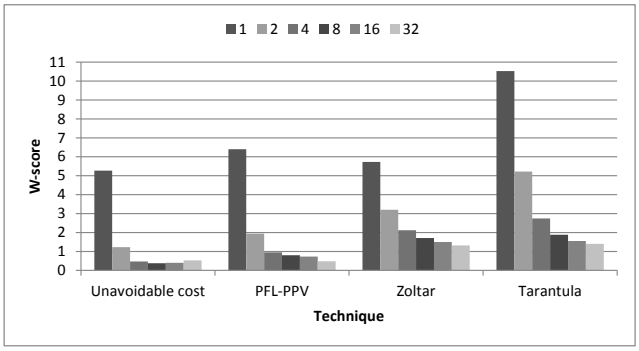

Figure 6.6: W-scores for selected techniques on Steimann benchmarks

is PFL-PPV, the top AVG A and W scores for the large benchmarks is also PFL-PPV. In addition PFL-SUPPES is only second to our automatically generated measure $m9185$ in terms of overall AVG W scores. Thus, our new techniques compare highly competitively.

We now discuss how PFL compares to SBFL in terms of overall efficiency. In our implementation it took under a second to find the most suspicious component in SBFL/PFL procedures. The complete PFL procedure, which recomputes the fault probabilities (as per the PFL-algorithm in section 6.6) until a fault is found, took slightly more than this – an average of 1.80 seconds – thus establishing PFL's negligible overhead for practical fault localisation purposes.

Note we can minimise runtime costs further if we only use the PFL-equations to

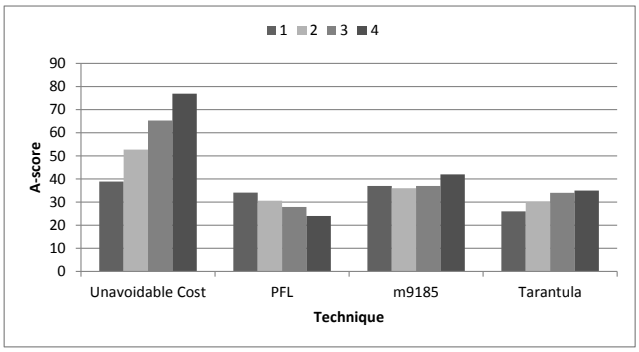

Figure 6.7: A-scores for selected techniques on SIR benchmarks

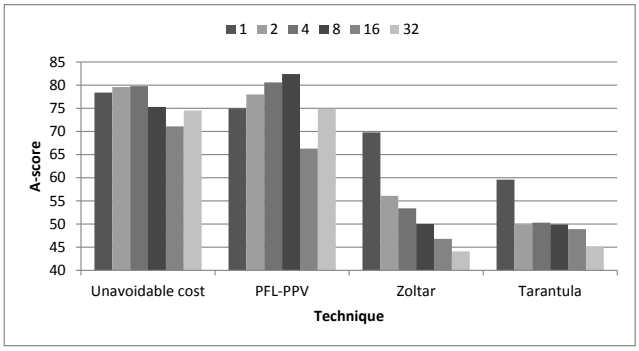

Figure 6.8: A-scores for selected techniques on Steimann benchmarks

determine the fault probability of each component once (forgoing iterative updates in probability), and this would conserve the A-score. This follows by definition, as our A-score measures whether the most probable fault is a fault.

We now discuss how choice of measure w makes a difference in overall effectiveness for the PFL framework. Choice in measure for PFL technique does not seem to make nearly as much difference in terms of overall effectiveness as it does in SBFL. To see this, compare PFL-PPV, PFL-SUPPES, PFL-OCHIAI, PFL-KULKZYNSKI2 with their counterparts PPV, Suppes, Ochiai, and Kulkzynski2 - the latter four have a substantially larger range in terms of W and A-scores when comparing the techniques scores on Tables 5.9, 5.10, and 6.2. This may suggest that the PFL-w framework can provides more robust fault localisation effectiveness irrespective of choice of measure.

We now discuss how the size of the program affects effectiveness for PFL techniques. To discuss this we compare the results of our PFL techniques for large (Steimann) and small (SIR) programs. We first discuss results for our small (SIR) benchmarks. PFL techniques were highly competitive (as Table 6.2 demonstrates). However, we noticed that 79.97% of the faults in the SIR benchmarks were covered by all failing traces which made it less challenging for our techniques in terms of noise. No established SBFL measure was statistically significantly better than our PFL techniques 40 A/W-scores on the SIR benchmarks. The experiments confirm PFL as high performing.

We now discuss results for our large (Steimann) benchmarks. For these benchmarks the PFL techniques performed better than SBFL techniques. Zoltar was the SBFL measure with the highest AVG W-score of 2.59. PFL-PPV improved on this score with a AVG W-score of 1.88. Thus, the user has to investigate 37.77% more code when using the best SBFL measure. Klosgen was the (established) SBFL measure with the highest AVG A-score of 55.2. (PFL-PPV) improved on this score by with a AVG A-score of 76.2. Thus, the user finds a fault immediately 27.56% less frequently using the next best established SBFL measures. Both PFL-PPV's W/A 60 scores were a statistically significant improvement over all SBFL measures using $p = 0.01$. Thus, the PFL approach was a substantial and significant improvement at localising faults on the large benchmarks.

Why are PFL advantaged on our larger benchmarks? PFL satisfies the fault localisation properties described in 6.5, meaning that the PFL framework makes components in failing traces with relatively fewer covered components relatively more suspicious. These failing traces were observed to be more abundant in the coverage matrices in our large benchmarks. An in-depth case study of a typical program version in the Steimann test suite which illustrates this point is given in the Appendix A.

Is this advantage important? We emphasise that performance in large programs such as the Steimann benchmarks is a highly desirable property. Thus far, only lightweight methods such as SBFL have been able to scale to benchmarks of this size, and industrial projects typical to the real world which require fault localisation methods are usually large.

We now discuss whether PFL experimentally outperforms the estimated upper bounds of SBFL effectiveness. We compare the unavoidable cost (UC) scores with our PFL-PPV for the large benchmarks. UC's AVG W/A-scores were 76.45 and 1.38 respectively. We compare this to PFL-PPV scores of 76.2 and 1.88 respectively. Thus, overall PFL was very similar to the estimated upper bounds of SBFL performance (as measured by UC scores). In particular, PFL-PPV outperformed UC's W-scores at

19/60 benchmarks (see Appendix A for these results), and UC's A-scores at 24/60 benchmarks. It is thus theoretically impossible to design a strictly rational SBFL measure that can outperform PFL-PPV on many of our large benchmarks. PFL-PPV techniques did not improve on UC scores on the small benchmarks.

We now discuss how measures behave as more faults are introduced into a program, as per Figures 6.6, 6.5, 6.8, 6.7. For SBFL measures a general trend is as follows: the more faults there were in a program the better an SBFL measure's W-scores, but the worse that measure's A-scores. In contrast, a similar negative trend for the A-scores was not noticed for our variants of PFL-w at our large benchmarks. This demonstrates a superior ability to deal with noise introduced by multiple faults in our large programs. The proposed explanation for this is twofold. Firstly, PFL satisfies the fault localisation properties described in 6.5, meaning that the PFL framework makes components in failing traces with few covered components relatively more suspicious (where such failing traces were observed to be more abundant in the large benchmarks). This in turn enables the framework to cut through noise which potentially confound SBFL techniques.

To ascertain the relationship between the number of faults in a program and the W-scores of PFL-PPV (described in Figures 6.6, 6.5, 6.8, 6.7), we used Spearman's rho. Here, the relationship between decreasing W-scores and increasing number of bugs was described as significant (R-score $<$ -0.99) for both the 4 SIR scores, and 6 Steimann scores. In addition, the relationship between decreasing A-scores and increasing number of bugs was described as significant (R-score $<$ -0.99) for only the SIR scores, but not the Steimann scores.

We now discuss how PFL behaves if we limit ourselves to investigating a certain percentage of the program. To discuss this, we appeal to the performance plot of Figure 6.3, which gives us a representation of how effective measures are if we limit ourselves to investigating n percent of a given program as averaged over all of our benchmarks. As we can see, performance clusters between PFL techniques, and is more or less similar to $m9185$ in overall performance. Small differences include an advantage of PFL-PPV when n is low, and a small advantage to $m9185$ when n is higher.

However, as PFL demonstrates superior ability at large programs, we focus our analysis on them by discussing Figure 6.4. For each set of n-fault benchmarks of the Steimann suite, if $y\,\%$ of the program versions received a W-score of $\leq x\,\%$, a point was plotted on that graph at (x, y). The mean (AVG) of the 6 graphs is also plotted. The figure demonstrates that if we limit fault localisation to only 10% of the blocks,

on AVG we would expect to find a fault 95% of the time using PFL-PPV. An outlier is that PFL-PPV does slightly worse on the 16-fault benchmarks. In general, the graph confirms the conclusion that PFL-PPV's performance is not substantially worsened by the number of faults in the program.

6.8 Summary

In this chapter we have presented a new formal framework of probabilistic fault localisation (PFL), and compared it to SBFL in terms of (1) desirable theoretical properties, (2) its effectiveness at fault localisation and (3) its efficiency. Regarding (1), the PFL equations were formally proven to satisfy desirable fault likelihood properties which SBFL measures could not. Regarding (2), PFL-PPV was shown to substantially and statistically significantly (using $p = 0.01$) outperform all known SBFL measures at W and A-scores on our large benchmarks (and remains competitive on small benchmarks). In particular we found an advantage on our large benchmarks: The user has to investigate over 37.77% more blocks of code (and finds a fault immediately 27.56% less frequently) than PFL-PPV when using the best SBFL measures. Furthermore, we showed that for many of our large benchmarks it is theoretically impossible to design strictly rational SBFL measures which outperforms PFL-PPV's W-scores. Regarding (3), we found that the PFL approach maintains a comparably negligible overhead to SBFL. Thus, our results suggest the PFL framework has theoretical and practical advantages over SBFL.

Chapter 7

Multiple-fault Localisation

In the previous two chapters we developed techniques designed to find a *single* fault in a program. However, as discussed in section 2.5, programs in the real world usually contain *multiple* faults [110, 166, 172]. Thus, there remains the problem of developing efficient and effective techniques for multiple fault localisation (MFL). An initial question is which of the following established strategies such techniques should adopt [235]:

1. *One-at-a-time strategy.* The engineer uses a given technique to localise a single fault – that fault is then repaired, the program is re-tested to check for further errors, and (if required) another round of single fault localisation is performed, with this process repeating until the desired number of faults are found.

2. *All-at-once strategy.* The engineer uses a technique to localise multiple faults in one sitting, resisting the requirement of the one-at-a-time strategy to re-execute the test suite after a single fault is found.

The first approach can be understood to reduce MFL to multiple iterations of single fault localisation. However, the major problem facing techniques which use this approach is that it may require many re-executions of test suites, which can take a prohibitively long time. Thus, in this chapter, our goal is to develop methods which can efficiently and effectively localise *multiple* faults using the potentially time saving "all at once" approach. As established in section 2.5, the major challenge facing many techniques with respect to this approach is that their effectiveness, as demonstrated in experimentation, has great potential for improvement. Following this challenge, our strategy is to develop a method which optimises given statistical techniques to

the task of MFL. The optimised techniques are then experimentally demonstrated to outperform state of the art statistical techniques at given MFL tasks.

The contributions of this chapter are summarised as follows:

1. We introduce and motivate a new algorithm M_g, which takes as input a given fault localisation method g (SBFL measure, PFL-PPV, or BARINEL), and optimises it for the purposes of multiple fault localisation using a greedy heuristic.

2. Building on Naish's notion of single fault optimal SBFL measures $Opt(g)$, we formally show that $M_{Opt(g)}$ satisfies a newly identified property of *multiple fault optimality*, and has a practical runtime comparable to the underlying technique being optimised.

3. To demonstrate the potential of $M_{Opt(g)}$ as a MFL optimiser, we experimentally show that the algorithm statistically significantly and substantially improves given SBFL fault localisation measures g at multiple fault-localisation tasks, and demonstrate $M_{Opt(g)}$'s negligible runtime when SBFL techniques are used as a value for g.

4. We find that $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$ perform best in experimentation – on average localizing over 3 faults after investigating 10% of the program blocks.

The rest of this chapter is organised as follows. In Section 7.1 we present a running example to motivate the task of MFL and demonstrate the need to improve existing SBFL methods. In Section 7.2, we propose a condition which describes when the search for faults should stop in instances of multiple fault localisation. In Section 7.3 we present our new algorithm. In Section 7.4 we identify a new property of multiple fault optimality and show that $M_{Opt(g)}$ satisfies it. Sections 7.5 and 7.6 provide our experimental results, which show how $M_{Opt(g)}$ can be used with existing ranking based techniques g at the task of MFL. Finally, we summarize our results in Section 7.7.

7.1 Motivating Example

We begin by presenting a small example in order to illustrate how SBFL could potentially be improved for the purposes of MFL. Consider the faulty `C` program `minmax2.c` in Figure 7.1. This program is the same as the one presented in Chapter 6, except that it has been modified to be a program with multiple faults, as opposed to

```
int main ()
{
  int inp1, inp2, inp3;
  int least = inp1;
  int most = inp1;

  if (most < inp2)
    most = inp2;  // C1

  if (most < inp3)
    most = inp3; // C2

  if (least > inp2)
    most = inp2; // C3 (fault#1)

  if (least > inp3)
    least = inp3; // C4

  if (least <= most)
     most = most + 1 // C5 (fault#2)

  assert(least <= most);
}
```

Figure 7.1: `minmax2.c`

one single fault. An error occurs just in case the assertion `least <= most` is violated. One explanation for the failure is the fault at `C3`, which should be an assignment to `least` instead of an assignment to `most`. A second explanation is the fault at `C5`, which should be modified in order to account for the possible overflow of the value of `most`. Thus, there are two faults which cause errors in the program. To generate a test suite for SBFL, we collected coverage data from ten test cases. Four of the test cases failed and six passed. The coverage matrix for the set of coverage vectors $\mathbf{T}$ for this test suite is given in Table 3.1. Accordingly, the program model for $\mathbf{T}$ is defined $\boldsymbol{PM} = \langle C_1, \ldots, C_5, E \rangle$, where $C_1 = \{t_4, t_5, t_{10}\}$, $C_2 = \{t_1, t_6\}$, $C_3 = \{t_1, t_2, t_3, t_8\}$, $C_4 = \{t_2, t_7, t_8, t_{10}\}$, $C_5 = \{t_4, \ldots, t_{10}\}$, and $E = \{t_1, \ldots, t_4\}$. By looking at the coverage matrix, we can observe that the two faults C_1 and C_3 can cause errors independently of each other (i.e. one can cause an error without the other being executed), and that no single fault is covered by all failing traces. These features contrast with the fault in `minmax.c` in Chapter 5. We may compute the program spectrum for each component using the coverage matrix as per usual. For example, the program spectrum for C_3 is $\langle 3, 1, 1, 5 \rangle$.

	C_1	C_2	C_3	C_4	C_5	E
t_1	0	1	1	0	0	1
t_2	0	0	1	1	0	1
t_3	0	0	1	0	0	1
t_4	1	0	0	0	1	1
t_5	1	0	0	0	1	0
t_6	0	1	0	0	1	0
t_7	0	0	0	1	1	0
t_8	0	0	1	1	1	0
t_9	0	0	0	0	1	0
t_{10}	1	0	0	1	1	0

Table 7.1: Coverage matrix for $\mathbf{T}$

We now illustrate how the SBFL algorithm is used to find both faults (as per [164]). We use the $Opt(\text{PPV})$ measure as an example. Accordingly, $Opt(\text{PPV})$ returns $a^i_{nf} + 2$ if $a^i_{ef} = |\mathbf{F}|$, and $a^i_{ef}/(a^i_{ef} + a^i_{ep})$ otherwise (see definitions of $Opt(g)$ in section 3.6, and PPV in section 5.1.2). Using this measure, the components $C_1, \ldots, C_5$ receive suspiciousness scores of 0.33, 0.5, 0.75, 0.25, 0.14 respectively, and thus the UUT corresponding to C_3 (fault #1) is inspected first by the user (as C_3 has the highest score of 0.75). However, C_5 (fault #2) is ranked the lowest with a score of 0.14. This means that if the user continues to investigate the UUTs in the program according to their descending order of suspiciousness, the user would have to investigate all of the program, which means that the fault localisation performed extremely poorly. In general, the example illustrates that SBFL methods can be very good at single fault localisation, but that there is potentially room for improvement for the purposes of multiple fault localisation.

7.2 When to stop searching for faults

In order to develop a new MFL method, we must first address the following question. That is, if the engineer will only stop the search for faults after multiple faults have been found, then what is the criterion for the engineer to stop searching for further faults? The proposal we defend is as follows.

- The search for faults stops when a faulty hitting set (FHS) has been found.

Faulty hitting sets were defined in the preliminaries (see section 3.2 of chapter 3). To our knowledge, this condition was first proposed by Gong et al. [87] in the context of SBFL. We call the condition *Gong's stopping condition*. Gong et al suggested that if one has found some faults which does not correspond to a faulty hitting set, then other faults must exist in the program, and thus the user should continue the search, because there must be some other program artefact in the program which is responsible for the remaining failing traces.

We now present additional arguments in favour for Gong's stopping condition. First, in terms of saving time for the fault localising engineer, the condition is potentially superior to the proposal of terminating the search after finding only a *single* fault. This is because finding and repairing a single fault one at a time, and then re-testing, can potentially be more time consuming than finding many faults and repairing them all at once, and then re-testing.

Second, Gong's criterion is potentially superior to the proposal of terminating the search only after *all* faults have been found. This is because in practice it is often unclear when this is the case, and so in the absence of a clear cause to stop the search, the user could continue indefinitely – potentially until all of the program has been investigated. Steimann et al. [213] argue this has further ramifications for the evaluation of fault localisation techniques in general, because "while the total number of faults will be known in an evaluation setting in which the faults themselves are known, in a practical application of fault localization it will not be (so that an evaluation should not be based on the localization of all existing faults)." We can summarise this argument as presenting an *epistemological problem* for the proposal that fault localisation methods halt after finding all faults – the user is not guaranteed knowledge of when this criterion has been satisfied.

Thirdly, even if we *do* know the exact number of faults in the program, finding all faults in one go (without repairing some of the faults and re-testing) can often be too ambitious. This is because the more faults there are in the program, the harder it can be to locate all of them even if the number of them is known owing to factors such as increased noise. This conclusion is confirmed by the studies of Jones et al. [64], Lucia et al. [163], and DiGiuseppe et al. [63]. We can summarise this second argument is as follows – even if we *do* know the exact number of faults in the program, finding all faults without at least repairing some of the faults and re-testing, can often be too ambitious.

Fourthly, given the premise that the set of faults must explain all failing traces, the total set of faults will have the property of being a faulty hitting set (FHS). This

means finding a FHS is a necessary condition of the project of completely debugging a given program if all faults are to be found in one go as per the *all-at-once* paradigm.

In summary, the proposal of finding a faulty hitting set in the context of multiple fault localisation, has the following advantages:

1. The user can locate more than one fault, potentially saving additional time spent on re-running a test suite.

2. The user will know when to terminate the search for faults in practice, if the number of faults is not known.

3. The user can potentially avoid continuing a search for additional faults when the data is extremely noisy and there are many faults in the program.

4. It provides us with an evaluation criteria in our experiments.

Thus, we conclude that finding a faulty hitting set in an all-at-once multiple fault localisation paradigm is a well motivated proposal, and we consequently adopt Gong's stopping condition as a criterion for the new algorithm.

7.3 MFL Algorithm

We now present our new algorithm, which we call M_g. In this notation M denotes the fact that a SBFL measure g is being optimised for locating multiple faults. M_g is a semi-automated procedure that guides the user to localizing a set of faulty UUTs which correspond to a faulty hitting set. The rest of this section is organised as follows. First we shall introduce some notation. Second, we shall outline and motivate our method. Third, we shall formally present the method itself. Fourth, we shall demonstrate how M_g can be used for improved fault localisation on the motivating example of section 7.1.

We begin with the notation. In order to keep the formal presentation of our method simple and high-level, it will be convenient to describe a semi-automated subroutine $localize(X, g)$, which describes a semi-automated procedure. Where X is an ordered set of components $\langle C_1, \ldots, E\rangle$, and g can be a SBFL suspiciousness measure (or PFL-PPV or BARINEL technique). When g is a SBFL measure, the subroutine is described as follows. Let $X = \langle C_1, \ldots, C_n, E\rangle$ First, each component $C_i \in X$ is associated with a spectrum $\langle a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np}\rangle$, where $a^i_{ef} = |C_i \cap E|$, $a^i_{nf} = |\overline{C_i} \cap E|$,

$a^i_{ep} = |C_i \cap \overline{E}|$, and $a^i_{np} = |\overline{C_i} \cap \overline{E}|$. g then assigns each component a suspiciousness score as a function of that components spectrum. Second, and following the SBFL method described in 3, the UUTs in the faulty program are investigated in descending order of suspiciousness until a fault is found. i is then returned by *localize* if the UUT corresponding to C_i is found to be faulty.

Additional assumptions underlying this process are as follows. First, the user will never investigate the same UUT twice in the practical investigation of the faulty program. Second, the user will always be able to identify a faulty UUT as faulty (and non-faulty UUT as non-faulty) if and when that UUT is investigated. Third, the user will only come to know whether a component is faulty (or non-faulty) by appeal to a call to *localize*. Fourthly, *localize* is only called when Gong's stopping condition has not yet been met (and thus when it is called, it may be assumed there still exists an undiscovered fault in the program to localize). For simplicity, we shall also use say that C_i is investigated just in case the UUT corresponding to C_i in the concrete faulty program is investigated. Finally, if *localize* is used as a subroutine for M_g, we shall assume the same measure g is being used on every call to *localize*, such that the suspiciousness score for a given component may be assumed be invariant according to M_g. Accordingly, this shall allow us to refer to the component's score as *the degree of suspiciousness using* M_g. Thus for each proband model if the first call to *localize* assigns C_i a suspiciousness score of x, it is always x, and the suspiciousness score of C_i is accordingly said to be x according to M_g.

Finally, we may also extend $localize(X, g)$ for when g is a method that takes as input a set of coverage vectors (such as PFL-PPV and BARINEL). Accordingly, for these techniques *localize(X, g)* proceeds as follows. First, a set of vectors is constructed for X. In a practical implementation, this is performed as follows. First, T is assigned a copy of $\bigcup X$, then for each $t_k \in T$ we re-assign the ith argument 1 if $t_k \in C_i$ and $C_i \in X$ (if it was not already 1 before). Fault localisation on $\langle X, T \rangle$ then takes place in the standard way associated with g, and an index to a newly found fault is returned to flow of control.

We now begin to informally motivate our semi-automatic fault localization procedure in order to give an intuition for the algorithm we propose. In order to motivate our method, we first describe a simple method of finding a faulty hitting set which we do not use. This is to generate all possible hitting sets, rank them in terms of suspiciousness, and then get the user to investigate UUTs corresponding to more suspicious hitting sets as a matter of higher priority until Gong's stopping criterion is met. However, given that the automatic stage of generating large quantities of hitting

sets can be potentially time-consuming, we offer a more lightweight method to find a faulty hitting set.

The method we propose is as follows. We assume the user is using a given fault localization method g and is given a proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ where $\boldsymbol{PM} = \langle C_1, \ldots, C_n, E \rangle$. At step 1, let $B = C_i \cup \ldots \cup C_k$ where $C_i, \ldots, C_k \in \boldsymbol{PM}$ and where and each of $C_i, \ldots, C_k$ have been found to be faulty by the user (here, we allow for the possibility that $B = \varnothing$). Now, if B is not a hitting set on $\mathbf{F}$, then the user must continue the search for faults as the stopping criterion has not yet been met. Now, instead of investigating the rest of the program immediately, the user constructs a new model X defined $\langle C_1 \cup B, \ldots, C_n \cup B, E \rangle$. The engineer then uses g to find the most suspicious faulty component in this new model (as per $localize(X, g)$). If the newly found faulty component in this new model is a faulty hitting set then the search terminates, otherwise the process re-iterates from step one. The key feature of the procedure is that each round of fault localisation is performed over components which are a superset of the union of faulty components found thus far. The practical benefit of this is that often faults which cover new failing traces (which old faults don't cover), can potentially be promoted in terms of suspiciousness – leading to quicker localization of faults and quick convergence to locating a FHS. This semi-automated process is formally presented as pseudo-code in Algorithm 2. As usual, $\mathbf{F}$ denotes the set of failing traces in $\mathbf{T}$. Further, for any ordered set X, the ith element of X is denoted $X[i]$. Comments to the pseudo-code are given in curly brackets.

Algorithm 2 *M_g - A greedy multiple fault localization procedure*

Input: A proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$, and measure g
Output: B, a faulty hitting set

1: $X \leftarrow \boldsymbol{PM}$
2: $B \leftarrow \{\}$
3: **while** $\mathbf{F} \subsetneq B$ **do**
4: **for** $0 < i < |X|$ **do**
5: $X[i] \leftarrow X[i] \cup B$
6: **end for**
7: $b \leftarrow localize(X, g)$ {semi-automated step}
8: $B \leftarrow B \cup \boldsymbol{PM}[b]$
9: **end while**
10: **return** B

Example 7.3.1. We illustrate how the algorithm can be used to find a faulty hitting set in the motivating example of section 7.1. Accordingly, $\mathbf{T}$ is represented by the coverage matrix of Table 7.1, $\boldsymbol{PM} = \langle C_1, \ldots, C_5, E \rangle$, g is $Opt(\text{PPV})$. We begin as follows. At line 1, X is assigned a copy of $\boldsymbol{PM}$ (X can be thought of as the program model used to find the next fault). At line 2, B is assigned the empty set (B can be thought of as the union of faulty components in $\boldsymbol{PM}$ found by M_g thus far). At line 3, the condition is true (as $\mathbf{F} \subsetneq \varnothing$ and $\mathbf{F} = \{t_1, t_2, t_3, t_4\}$. Here the test is whether a faulty hitting set has been found yet). At lines 4 to 6, for $0 < i < |X|$, each ith element of X ($X[i]$) is assigned $C_i \cup \varnothing$ (in future iterations, this loop will update the program model with the last found found). Thus $X = \boldsymbol{PM}$ by line 7. At line 7, the semi-automated *localize* sub-routine is called, in which SBFL commences using $Opt(\text{PPV})$ and X, with the components $C_1, \ldots, C_5$ receiving suspiciousness scores of 0.33, 0.5, 0.75, 0.25, 0.14 respectively (note that this is exactly the same SBFL process as described in the motivating example of section 7.1). As C_3 is found to be the faulty component by the user, the index 3 is assigned to b at line 7. At line 8, $\varnothing \cup \boldsymbol{PM}[3]$ (which is equal to C_3) is assigned to B (thus C_3 is fault found thus far).

Flow of control returns to line 3, where the condition $\mathbf{F} \subsetneq C_3$ is true (as $\mathbf{F} = \{t_1, t_2, t_3, t_4\}$ and $C_3 = \{t_1, t_2, t_3, t_8\}$, and thus a faulty hitting set has not yet been found). At lines 4 to 6, for each $0 < i < |X|$, the ith element of X ($X[i]$) is assigned $X[i] \cup C_3$. Thus, if $X = \langle D_1, \ldots, D_n, E \rangle$, then for each $0 < i < |X|$ we have $D_i = C_i \cup C_3$ (here, the program model is updated with the last found fault). At line 7, the semi-automated *localize* sub-routine is called, and SBFL then commences using $Opt(\text{PPV})$ with X, with the components $C_1 \cup B, \ldots, C_5 \cup B \in X$ receiving

suspiciousness scores of 5, 0.6, 0.75, 0.5, 2 respectively. Accordingly, the user inspects the UUT corresponding to C_1 first (not faulty), then that of C_5 (faulty), and so b is assigned 5. At line 8, we then have $B \leftarrow C_3 \cup C_5$ (as the faulty components found thus far are C_3 and C_5). Flow of control returns to line 3, where the condition is found to be false (as $\mathbf{F} \subseteq C_3 \cup C_5$ and $C_3 \cup C_5 = \{t_1, \ldots, t_{10}\}$). $C_3 \cup C_5$ therefore satisfies the condition of being a faulty hitting set. The set is returned to the user and the algorithm terminates. Overall, this example illustrates that using $Opt(\text{PPV})$ as a value for g in M_g can be an improvement over using $Opt(\text{PPV})$ alone – in the motivating example all non-faulty UTTs need to be investigated before all faults are found, whereas now only 1 non-faulty component was investigated.

7.4 Properties

In this section we first present a new formal property of multiple fault optimality. We then argue this property is desirable for any multiple fault localisation method to satisfy, and thirdly we prove that $M_{Opt(g)}$ satisfies this property. Finally, we discuss both theoretical and practical advantages of $M_{Opt(g)}$.

We begin by stating the new property. For a given program model $\boldsymbol{PM} = \langle C_1, \ldots, C_m, E \rangle$, then

Definition 7.4.1. MULTIPLE FAULT OPTIMALITY. If there are $n+1$ faulty components, and only n known faulty components $B = C_s \cup \ldots \cup C_t$, then $(\forall C_j, C_k \in \boldsymbol{PM})$ if $\mathbf{F} \subseteq C_j \cup B$ and $\mathbf{F} \nsubseteq C_k \cup B$ then $C_j \cup B$ is more suspicious than $C_k \cup B$.

We hold that any multiple fault-localisation method must satisfy this property in order to be optimal. The observation that it is an optimal property is grounded in the assumption that the union of faulty components must cover all failing traces. That is, if there are n faulty components in the program model, and $n-1$ are known to be faulty, and there are some failing traces which do not cover the union of known faulty components, then it must be the case that the remaining failing traces cover a single unknown faulty component. The new property is similar to Naish's property of single fault optimality discussed in Section 3.6, which implies that components which cover all failing traces should be investigated first, under the assumption there is only one faulty component in the program model. Here the property is extended to multiple fault cases. We now prove that $M_{Opt(g)}$ satisfies single and multiple fault optimality. As follows:

Proposition 5. $M_{Opt(g)}$ satisfies properties of single and multiple fault optimality.

Proof. We first do the case for single fault optimality. To show that $M_{Opt(g)}$ is single fault optimal, we must show that for the given proband model $\langle \boldsymbol{PM}, \mathbf{T} \rangle$, and for all $C_i, C_j \in \boldsymbol{PM}$, if $a^i_{ef} = |\mathbf{F}|$ and $a^j_{ef} < |\mathbf{F}|$ (or $a^i_{ef} = a^j_{ef} = |\mathbf{F}|$ and $a^i_{ep} < a^j_{ep}$) then C_i is more suspicious than C_j using $M_{Opt(g)}$ (by the definition of single fault optimality of section 3.6). This obtains as $Opt(g)$ is single fault optimal by definition.

It remains to show that $M_{Opt(g)}$ satisfies the property of multiple fault optimality. We show that for a given program model $\boldsymbol{PM} = \langle C_1, \ldots, C_n, E \rangle$ the consequent of the conditional of the property is trivially satisfied by $M_{Opt(g)}$. Assume there are n (possibly 0) known faulty components $B = C_s \cup \cdots \cup C_t$ where $C_s \cup \cdots \cup C_t \in \boldsymbol{PM}$, such that $\mathbf{F} \subsetneq B$. We assume that these faults have been found using iterative calls to *localize* (as per our assumptions in section 7.3). Accordingly, the inner for loop constructs a program model $X = \langle C_1 \cup B, \ldots, C_n \cup B, E \rangle$. Now, using the single bug optimal method $Opt(g)$ to rank each component in X, we have the following for each $C_i, C_k \in \boldsymbol{PM}$: if $\mathbf{F} \subseteq C_i \cup B$ and $\mathbf{F} \subsetneq C_k \cup B$ then $C_i \cup B$ is more suspicious than $C_k \cup B$ (as per the definition of $Opt(g)$). Thus $M_{Opt(g)}$ satisfies single and multiple optimality.

□

We note that the reason $M_{Opt(g)}$ satisfies the property of multiple fault optimality is because it assumes that the antecedent of the conditional is trivially fulfilled. Explicitly, $M_{Opt(g)}$ always works on the assumption that if n faults have been found which do not cover all failing traces, then there exists only one extra fault in the program that needs to be found. In this sense the underlying heuristic of the algorithm is provided for by *Occam's razor*. Occam's razor states that of competing hypotheses the simplest is to be preferred. In this case, the simplest hypothesis is the one which assumes the smallest number of faults remaining – one. Thus, the underlying assumption is similar to Naish's single bug assumption behind the use of single bug optimal measures, which is the (stronger) assumption that there is only one fault in the program [181].

We now discuss the potential theoretical and practical advantages of satisfying the new property. First, the theoretical advantages. In general, $M_{Opt(g)}$ has the potential to improve the fault localisation *effectiveness* (measures in terms of W-scores) of SBFL measures by satisfying both properties of single and multiple fault optimality (which measures do not automatically satisfy by themselves). First, as $M_{Opt(g)}$ is single fault optimal, it will outperform non single-fault optimal measures for cases when there is

one fault in the program and none have yet been found. Second, as $M_{Opt(g)}$ is multi fault optimal, it will outperform non-optimised measures for cases when there are $n+1$ faults in the program and n have been found. Finally, it is observed that the algorithm has the potential to be efficient in practice, if an efficient technique is used as the underlying fault localisation method (g) and the number of faults found by the algorithm (n) is sufficiently small. This is because the runtime will be comparable to performing n iterations of the underlying technique used.

Second, the practical advantages. In practice, bugs in programs are assumed to be quite sparse. This is at least the case for our benchmarks, and is understood to be generally in the case in projects under test in the real world. Thus, supposing there are few faults in the program, we can often presume that a simple hypothesis (i.e. that there are a small number of bugs) is correct, and thus the assumption of Occam's razor underlying our approach is potentially often applicable in practice.

7.5 Empirical Evaluation 1: Optimised SBFL and PFL

In this section we present results for our first multiple fault localisation task. The task of the experiment is to determine how effective and efficient $M_{Opt(g)}$ is at finding a faulty hitting set when g is a SBFL method is used, and where we shall also extend our implementation of $M_{Opt(g)}$ to include PFL-PPV as a value for g. The experimental setup is presented in 7.5.1, we then present our experimental results in sections 7.5.2 and 7.5.3. Finally, we discuss these results in section 7.5.4.

7.5.1 Setup

We evaluated techniques on their ability to localise a faulty hitting set (FHS). We included three statistics to assess a technique: a wasted effort score (W-score), the number of faults isolated by the technique, and a score which measures the fault localisation rate of a given technique (rate score). We also used Wilcoxon rank sum tests to determine whether one method's scores was significantly better than another ($p = 0.01$). We describe these below.

We begin with the *wasted effort scores*. For a given technique we wish to calculate the percentage of non-faulty components we'd expect a user to investigate until a faulty hitting set is found. The subtlety here was how to score in cases where a most

suspicious fault is tied. The major difficulty here is how to assess $M_{Opt(g)}$ efficiently. To see the difficulty, we envisage scenarios in which $M_{Opt(g)}$ is being used, and $Opt(g)$ ranks several different faults with a highest (equal) degree of suspiciousness, such that if the user chooses to investigate any one of these, then in the next round of fault localisation the engineer is faced with an analogous scenario – in which the engineer must again choose to investigate one of many equally ranked faults with a highest degree of suspiciousness (with this pattern potentially repeating many times). Now, to calculate the average wasted effort we'd need to know each series of choices the user could make in this fashion until successfully localizing a faulty hitting set. However, given the number of series of choices can grow exponentially on the number of faults in the program (there is a maximum of 32 in our experiments), making this calculation can have high overhead, as has been confirmed by experimentation.

Consequently, we had to use a different definition of a wasted effort score which was more tractable. This was, in the case of ties, that the user chose one of the tied components at random. This meant that every suspiciousness list for every technique became a strict order, and our wasted effort score was then defined as the percentage of non faulty program components investigated by the user until a faulty hitting set was found. Formally, our wasted effort score was as follows. Let h be the number of non-faulty components investigated until FHS is found, and let $\mathbf{PM}$ be the program model under test (including all components except the error E), then:

$$W = \frac{h}{|\mathbf{PM}|}100 \tag{7.1}$$

This scoring method is an adapted version of the *worst* case W-score used for assessing techniques at single fault localisation (see Section 4.2). Given our granularity of program component were blocks (which puts lines of the code with the same coverage properties in the same block, see Section 4.2), ties were rare, and in re-runs of experiments this randomisation did not have any observable effect on AVG scores (reported to two decimal places).

We now discuss the potential of using *absolute scores* for the task of faulty hitting set localisation. We note that if one is engaging in user-guided SBFL of multiple bugs, then limiting oneself (following Parnin and Orso [188]) to investigating only a small number of highly suspiciousness components is inappropriate. This is because in some cases we are searching for large numbers of bugs (e.g. 8, 16, 32), which will greatly exceed the small number of components a user is permitted to investigate under this evaluation. Thus, we conclude that A-scores are not as important for the assessment of method effectiveness for the MFL task, and consequently do not use them.

A second means of assessment is the number of faults isolated. Thus, in our results we present the mean number of faults isolated for the set of n-fault benchmarks in a given suite, and the mean number of faults localised overall for each suite (Steimann and SIR), and the overall mean number of faults localised overall (the mean of the 100 n-fault benchmarks). A third means of assessment is the fault localisation *rate*. For a given program version the fault localisation rate is the number of components investigated (as measured by the W-score) divided by the number of faults in the FHS found. This number gives the average number of blocks investigated for every fault found. In our results we present the mean rate for the set of n-fault benchmarks in a given suite, (Steimann and SIR), and the overall mean rate overall (the mean of the 100 n-fault benchmarks). Finally, we used Wilcoxon rank sum test to determine whether one technique's 100 n-fault benchmark W-scores were significantly lower (better) than another. We think W-scores are the most important scoring method, as we want our techniques to isolate a faulty hitting set by the user investigating as few program blocks as possible.

We now discuss SBFL measures compared. Large scale experimentation over $M_{Opt(g)}$ techniques is more time costly, as for every technique g we have to independently and iteratively compute a series of test suites for each corresponding fault localisation technique. Thus, it was impossible to investigate every SBFL measure g with an optimised version M_g or $M_{Opt(g)}$ due to the excessive amount of time the potential experiment would take. Therefore a selection of SBFL measures were chosen based on high performance of g at the task of single fault localisation in Chapter 5. These included the following measures: $m9185$, Lewis, Popper2, Kulczynski2, Ochiai, and PattSim. We additionally chose Jaccard, Tarantula, and YulesQ to examine how worse performing measures would be optimised.

7.5.2 Scores by average

We now present the AVG W-scores of the selected techniques in Table 7.2. The first column gives the name of the technique g. The rest of the table is presented in three sets of three columns. The first set give AVG scores for all n-fault benchmarks (all 100), the next three gives AVG results for all n-fault Steimann (STI) benchmarks (all 60), and the next three for all n-fault SIR benchmarks (all 40). To indicate how much $M_{Opt(g)}$ improves over $M_{(g)}$ and g, we present the last two columns which give overall AVG W-scores. Notation is as follows. **w** gives the mean W-score, **bf** is the mean number of bugs found, **rate** gives the mean fault localisation rate. In

g	AVG $M_{Opt(g)}$ **w**	**bf**	**rate**	AVG (STI) **w**	**bf**	**rate**	AVG (SIR) **w**	**bf**	**rate**	AVG M_g **w**	g **w**
PFL-PPV	**10.00**	3.69	2.72	**8.06**	5.19	1.56	12.90	1.42	9.08	-	-
Lewis	**10.00**	3.79	2.64	8.25	5.37	1.55	**12.63**	1.42	8.91	15.63	29.69
Popper2	12.03	3.91	3.08	11.43	5.56	2.06	12.93	1.43	9.02	23.99	30.85
K2	12.59	3.94	3.20	12.27	5.61	2.19	13.07	1.42	9.20	21.37	27.99
Ochiai	12.67	3.93	3.23	12.53	5.59	2.24	12.88	1.43	9.02	26.36	31.51
Jaccard	13.06	3.93	3.32	12.99	5.60	2.32	13.17	1.43	9.19	29.55	35.03
YulesQ	13.94	4.19	3.32	14.43	6.04	2.39	13.20	1.42	9.28	23.21	30.85
m9185	14.51	4.12	3.53	15.36	5.91	2.60	13.23	1.42	9.32	25.37	28.92
Pattsim	15.34	4.24	3.62	16.18	6.11	2.65	14.08	1.43	9.82	18.89	30.63
Tarantula	19.18	4.44	4.32	21.79	6.43	3.39	15.27	1.45	10.51	40.93	36.21

Table 7.2: MFL AVG scores

Appendix A.3 we provide tables which describe the 100 scores for individual n-fault benchmarks for our top performing techniques for the interested reader.

7.5.3 Scores by number of faults

In this section we present four tables to describe results as a function of the number of faults in a benchmark. Figures 7.2 and 7.3 are presented in the same way as in previous chapters: The y-axis gives the W-score, the x-axis groups results for each technique, with the number of faults in the benchmark represented by the shade presented in the legend (For instance, Ochiai has a W-score of over 50% for the 16 fault versions of the Steimann benchmarks). Figures 7.4 and 7.5 plot the average number of faults found per n-fault benchmark in each of the small (SIR). The first set of columns is *covered bugs*, which represents the average number of covered faults per n-fault benchmark. The remaining columns give the average number of faults found by that technique for the set of n fault benchmarks in that suite. The legend gives the number of bugs by shade. We chose to represent the particular measures in the figures as they demonstrated a range of performance.

7.5.4 Discussion

We now discuss our results. Our discussion is designed to ascertain the performance of $M_{Opt(g)}$ as an effective and efficient method of fault localisation.

We first discuss how effective $M_{Opt(g)}$ techniques were in terms of W-scores. To discuss this by appealing to the results of Table 7.2. We first discuss g and M_g tech-

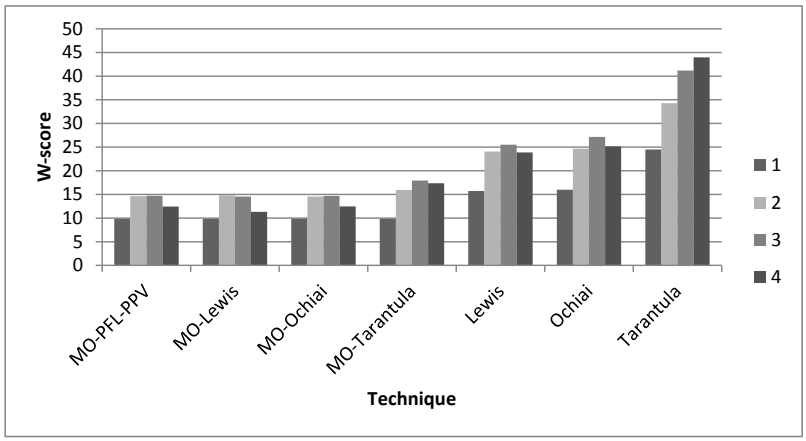

Figure 7.2: W-scores for selected techniques on SIR Benchmarks

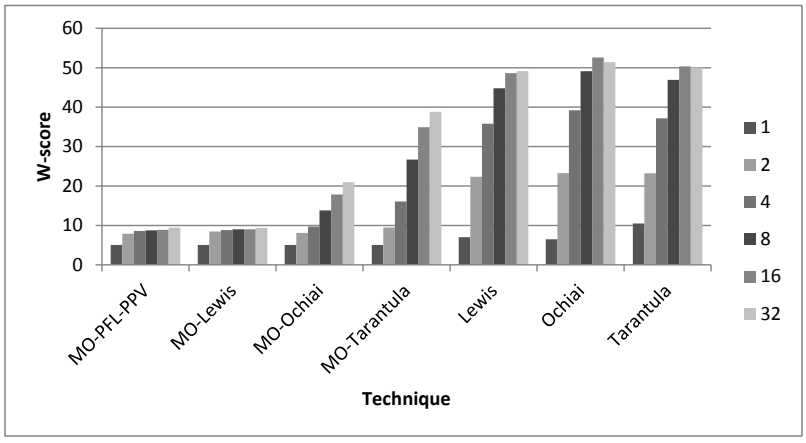

Figure 7.3: W-scores for selected techniques on Steimann Benchmarks

niques. We observed that optimising a measure g using M_g provided a slight (but statistically significant) improvement on the scores of g. The single exception was Tarantula, which provided for a small decrease in performance. Unoptimised measures g were relatively ineffective. For instance, the range in W-score was 27.94% to

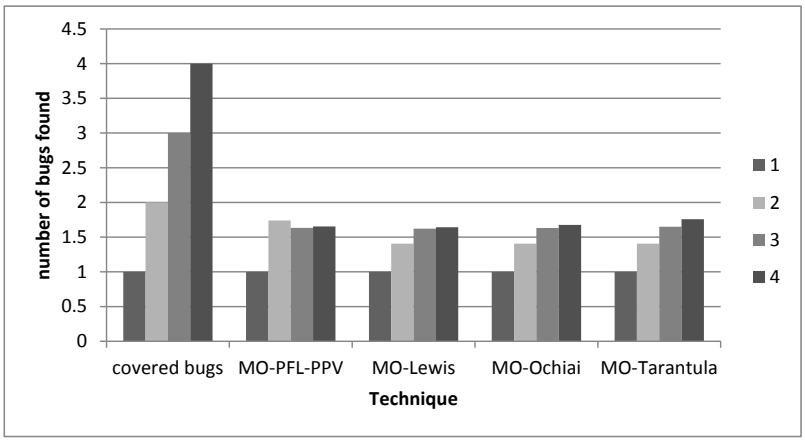

Figure 7.4: Number of faults found on SIR Benchmarks

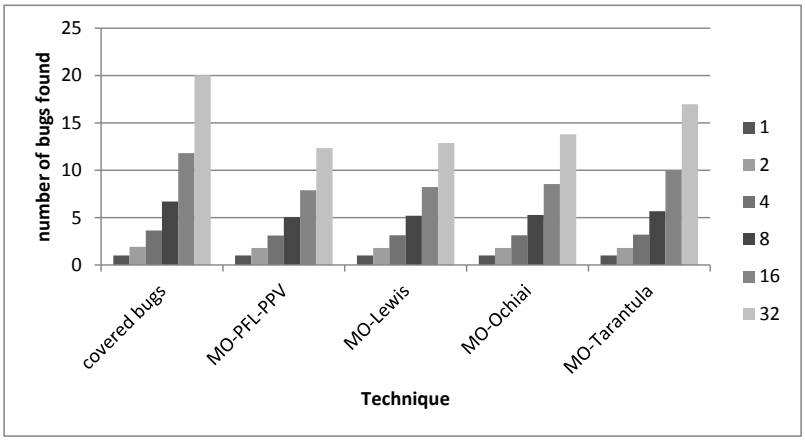

Figure 7.5: Number of faults found on Steimann Benchmarks

45.38% when all unoptimised measures (presented in Table 5.9) were tested. This confirms existing experimental results that SBFL measures are not effective as standalone multiple fault localisation techniques [164].

We now discuss $M_{Opt(g)}$ techniques. In general, improvement of W-score can be observed when we used $M_{Opt(g)}$ as opposed to g or M_g (compare the last two columns

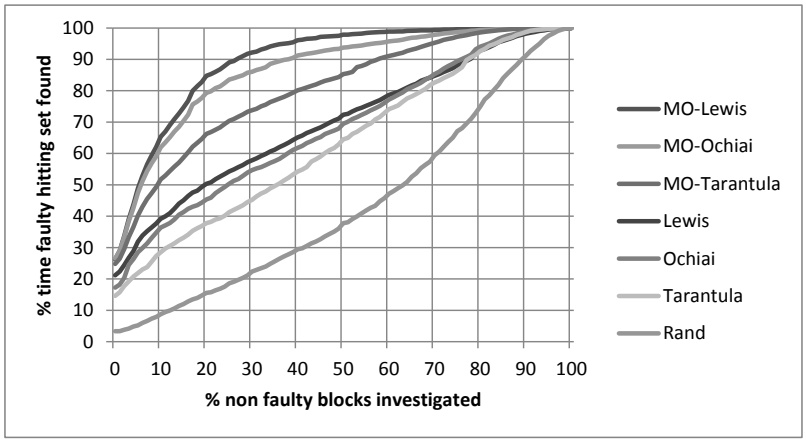

Figure 7.6: Performance of SBFL techniques optimised and unoptimised

of the table with the first column). The improvement is statistically significant in all cases where an SBFL measure g is optimised. The improvement was also substantial, in general using a SBFL measure g to find a faulty hitting set often required the user to inspect two to three times more blocks of code. Thus, $M_{Opt(g)}$ improves in effectiveness over unoptimised SBFL techniques g in terms of W-scores.

To emphasise the effectiveness of $M_{Opt(g)}$ as an optimiser of SBFL techniques, we observe the best performers were $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$, which on AVG localised a faulty hitting set with a W-score of 10.00%, and statistically significantly outperformed all techniques g in Table 7.2. These techniques found faulty hitting sets (consisting of an average of 3.79 and 3.69 faults respectively) with almost the same W-score as the Tarantula measure when it finds a single fault (compare the overall W-scores of 10.00 for $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$ at the multiple fault localisation task and 9.53 for Tarantula at the single fault localisation task – see Table of W-scores in Chapter 5). Thus, the prospect of finding hitting sets of multiple faults is as plausible a proposition as single fault localisation is for Tarantula.

We now discuss how effective $M_{Opt(g)}$ techniques were in terms of the number of faults found. The average number of faults found per n fault benchmark in each of the small (SIR) and large (Steimann) suites for a range of $M_{Opt(g)}$ techniques are presented in Figures 7.4 and 7.5. Three were three general trends:

1. The more faults there were in a program the more faults were isolated.

2. The more faults there were in a program the smaller the proportion of the faults in the program found.

3. The better the W-score of an optimised technique the fewer faults were found by it.

We now discuss these trends. Concerning the first trend – for the SIR benchmarks there was only a small increase in number of faults isolated as the number of faults increased. We discovered this was because almost 80% of the faults in the SIR benchmarks were in fact faulty hitting sets, which meant that if the algorithm found one fault the algorithm would terminate early, as a faulty hitting set would have been found. The SIR benchmarks have this property because many of the programs are quite small and have less branches than the larger Steimann benchmarks. In contrast, for the Steimann benchmarks the trend was more accentuated. As more faults were introduced into the program, the more faults were isolated. This confirms that $M_{Opt(g)}$ is effective at finding many faults on our large programs.

Concerning the second trend – to illustrate this trend we can review the data for our top performer $M_{Opt(Lewis)}$ in terms of proportions (rather than number) of faults found. This measure located 75.74% of the faults overall (an average of 3.79 bugs located in programs with an average of 5.51 covered bugs overall). In the SIR benchmarks, it located 100% of the faults in the 1-fault versions, and 70.3, 54.1, 41.13 % in the 2, 3, 4 fault versions respectively. In the Stiemann benchmarks, it located 100% of the faults in the 1-fault versions, and 93.54, 86.49, 77.86, 69.85, 64.11% in the 2, 4, 8, 16, 32 fault versions respectively. Thus, as more faults are introduced, a smaller proportion of faults are found. This trend was representative for all our optimised measures. Further details of $M_{Opt(Lewis)}$ are presented in Appendix A.3. Consequently, we believe $M_{Opt(g)}$ is effective in terms of number and proportion of faults isolated.

Concerning the third trend – this represented a trade-off between how quickly the technique terminated and how many faults were found. This trade-off was slight, as all optimised measures localised between 74% and 79% of the covered bugs.

We conclude that our $M_{Opt(g)}$ techniques effectively localise a substantial proportion and number of bugs.

We now discuss how choice in SBFL measure g affected $M_{Opt(g)}$'s W-score. Using different SBFL measures g with $M_{Opt(g)}$ presents a substantial range in performance

as more faults are introduced into a program. For instance, Figure 7.3 shows that using Ochiai as a value for g in $M_{Opt(g)}$ greatly improved upon Tarantula, and Lewis greatly upon Ochiai. Furthermore, Lewis' performance is relatively invariant the more faults are introduced into the program – where this is not the case for Ochiai or Tarantula. This suggests that choice of SBFL measure can make a substantial difference in effectiveness as the number of faults in the program increases.

We now discuss the efficiency of $M_{Opt(g)}$ techniques in practice. In theory $M_{Opt(g)}$'s complexity is comparable to performing multiple iterations of the underlying technique g. $M_{Opt(Lewis)}$ took an average of 0.60 seconds per program version, $M_{Opt(pfl-ppv)}$ was comparable with an average of 0.69. This establishes the approach as efficient for use with SBFL and PFL techniques.

We observed that the runtime for $M_{Opt(pfl-ppv)}$ when used for the multiple fault localisation task was less than when PFL-PPV was used for the single fault localisation task (which was 1.80 seconds on average). This was due to the fact that our implementation of $M_{Opt(g}$ automatically ranks components covered by all failing traces as most suspicious, thereby circumventing the requirement for running the full PFL-algorithm.

We now discuss how the introduction of more faults affected the effectiveness of $M_{Opt(g)}$ techniques. To discuss this we discuss Figures 7.2 and 7.3, which are constructed with the same methodology as the analogous figures of Section 5.3 The main observation was as follows: The more faults in a program, the less effective the technique was in finding a faulty hitting set (measured in terms of W-score). The observed explanation for this is that the faulty hitting sets located tended to be larger in the larger programs, requiring more time to find. We also observe in the figures that the difference in W-score between $M_{Opt(g)}$ and g techniques increases the more faults there are in a program in terms of W-scores – especially for our large benchmarks.

To ascertain the relationship between the number of faults in a program and the W-scores of $M_{Opt(g)}$ when g = PFL-PPV (as described in Figures 7.2 and 7.3), we used Spearman's rho. Here, the relationship between increasing W-scores and increasing number of bugs was described as significant (R-score $<$ -0.99) for the 6 Steimann scores, but not the 4 SIR scores.

We now discuss how limiting oneself to investigating a certain percentage of the program affected effectiveness of $M_{Opt(g)}$ techniques. We discuss this with appeal to Figure 7.6, which demonstrates the performance of optimised and unoptimised versions of Lewis, Ochiai, and Tarantula for locating a faulty hitting set. The measures

selected in the figure were chosen to represent a range of performance. The method for constructing the figure was analogous to the similar figures in Section 5.3. Note that when the percentage of non-faulty blocks investigated is set to zero, we have the absolute scores for the given technique. The curve for $M_{Opt(pfl-ppv)}$ is almost identical to $M_{Opt(Lewis)}$ and thus was not presented. Thus, $M_{Opt(Lewis)}$, $M_{Opt(Ochiai)}$, $M_{Opt(Tarantula)}$, Lewis, Ochiai, Tarantula, and Rand had absolute scores of 26.24, 26.52, 24.78, 21.17, 17.25, 14.6, and 3.34% respectively. These are not as good as the absolute scores for the single fault localisation task, which is to be expected as the problem is harder. Figure 7.6 demonstrates that optimisation makes a substantial difference in overall performance compared to unoptimised measures.

In summary, our optimising algorithm $M_{Opt(g)}$ is effective and efficient at fault localisation when used with Lewis and PFL-PPV techniques. Moreover, they are a substantial and statistically significant improvement on the standalone SBFL measures compared in terms of W-scores.

7.6 Empirical Evaluation 2: Optimised Barinel

Barinel has been demonstrated to be an effective multiple fault localisation tool at the state of the art of statistical fault localisation [13,16,17,20,21]. Thus, we assume that if (for some suitably chosen g) $M_{Opt(g)}$ can shown to improve on Barinel's W-scores in large scale experimentation, then this also demonstrates the effectiveness of the new method.

Experiments for Barinel using $M_{Opt(g)}$ could not be completed for all of our program versions in our benchmarks in a reasonable time frame for the purposes of this dissertation. This was because Barinel is a significantly more heavyweight tool than SBFL or PFL technique, often requiring many minutes to perform fault localisation tasks. To illustrate how such a tool cannot scale to our experiments, we calculated that techniques which take 10 minutes on average would take almost a year to complete all 50K+ program versions in our benchmarks. Thus, in this section we present a smaller scale experiment, results of which demonstrate whether the Barinel tool could be optimised by $M_{Opt(g)}$, and/or perform competitively with our best performing $M_{Opt(g)}$ techniques of the previous section: $M_{Opt(Lewis)}$ or $M_{Opt(pfl-ppv)}$. In section 7.6.1 we present our experimental setup, and then in section 7.6.2 present and discuss our results.

7.6.1 Setup

We first describe the Barinel procedure [13,17,20,21]. Firstly, a minimal hitting set algorithm Staccato is used to generate a set of hitting sets (aka hypotheses/diagnoses) – in their experiments and ours a limit of 100 hitting sets is chosen. Secondly the ranking component of Barinel uses advanced Bayesian methods to rank each hitting set in terms of suspiciousness. Thirdly, the most suspicious hypothesis is presented to the user for inspection, and each component in it is inspected as to whether it is faulty/healthy. Fourthly, fault probabilities are updated after each hitting set is inspected and the algorithm reiterates until the desired number of faults are found. Barinel is at the state-of-the-art with respect to statistical multiple-fault localisation. As the full algorithm is relatively sophisticated, we refer the reader to Abreu [13, 17, 20, 21] for full details. Implementations of the versions of Staccato and Barinel used in our experiments were gratefully obtained from the authors via personal correspondence.

We now discuss modifications to Barinel for the purposes of our experiment. We found certain modifications to Barinel improved its W-scores for the multiple-fault localisation task. In the latest implementation of Barinel there is an option which returns the most suspicious component of the most suspicious hypothesis, and we found that replacing step three of the procedure described above with a step which only returned *one* component (as opposed to an entire hitting set) to the user for inspection substantially improved results on the experiments in this section. This was observed to be because the fault probabilities received an immediate update after every single inspection of a program component, as opposed to after inspection of all the program components in the hitting set. As the results for this modified version were superior in terms of W-scores, we used the modified version in our experiments.

We now discuss the programs used for our experiments. For our experiments we used all of the SIR benchmarks, along with 20 randomly chosen program versions from each of 1/2/4/8 benchmarks for each program (e.g. 20 versions of the 1 fault Daikon, 20 versions of the 2 fault Daikon, etc.) making for a total of 800 program versions from the Steimann benchmarks. We had a 20 minute time-limit for Barinel. We could not get Barinel to work efficiently enough for a meaningful comparison of 16 or 32 fault benchmarks. This was because their implementation of the ranking component of Barinel could not process hitting sets much larger than 16 components, which led to timeouts. We are aware that next generation implementation of their tool may overcome this difficulty.

7.6.2 Results

We summarise the results for our smaller scale experiment in this section. The first result is that BARINEL does not perform competitively in terms of efficiency, and is slightly worse in terms of effectiveness, than $M_{opt(g)}$ when g is Lewis or PFL-PPV. The second major result is that $M_{opt(Barinel)}$ is a substantial improvement over Barinel. We summarise results with the following discussion.

We first discuss how effective the techniques were in terms of W-scores. BARINEL received an AVG W-score of 17.32. The performances of $M_{Opt(pfl-ppv)}$, $M_{Opt(Lewis)}$, $M_{Opt(Ochiai)}$, and $M_{Opt(Barinel)}$ substantially and statistically significantly improved on BARINEL's W-scores (scoring 10.10, 10.11, 10.84, 10.84 respectively overall), with neither significantly outperforming the other. Thus, $M_{Opt(Barinel)}$ is an improvement on BARINEL. $M_{Opt(pfl-ppv)}$ was the top performer with a score of 10.10.

We now discuss how effective the techniques were in terms of number of faults localised. Results were not particularly different between approaches, and so we briefly present the results here. $M_{Opt(Lewis)}$, $M_{Opt(Barinel)}$ and BARINEL, $M_{Opt(pfl-ppv)}$ localised an average of 2.18, 2.07, 2.16, 2.09 faults respectively. Thus $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$ slightly outperformed $M_{Opt(Barinel)}$ in terms of number of bugs localised.

We summarise the results in greater depth here. For the 1/2/3/4 fault versions of the SIR benchmarks $M_{Opt(Barinel)}$ localised 1.00, 1.41, 1.64, and 1.70 faults on average. For the 1/2/4/8 fault versions of the Steimann benchmarks it localised 1.00, 1.79, 3.17, and 4.90 faults on average. This was comparable to $M_{Opt(Lewis)}$, which obtained the results 1.00, 1.41, 1.62, and 1.65, for SIR (repeating the results of the MFL experiment in the previous section), and 1.00, 1.80, 3.19, 5.24 faults on average for the Steimann benchmarks. This, again, was comparable to $M_{Opt(pfl-ppv)}$, which obtained the results 1.00, 1.41, 1.63, and 1.55, for SIR, and 1.00, 1.80, 3.16, 5.08, faults on average for the Steimann benchmarks. Thus, the number of faults found were very similar.

We now discuss how $M_{Opt(Barinel)}$ performed as more faults are introduced in the program. To discuss this we appeal to Figures 7.7 and 7.8. As we can see, $M_{Opt(Barinel)}$ substantially improves the performance of BARINEL for each class of programs with n-faults. However, all optimised techniques performed similarly in terms of W-scores. The general trend is that the more faults are introduced, the worse the W-score. An outlier to this trend is observed for BARINEL for two fault programs in the Steimann benchmarks as portrayed in Figure 7.8. We discovered that these results were a consequence of the fact that in many of the two fault versions

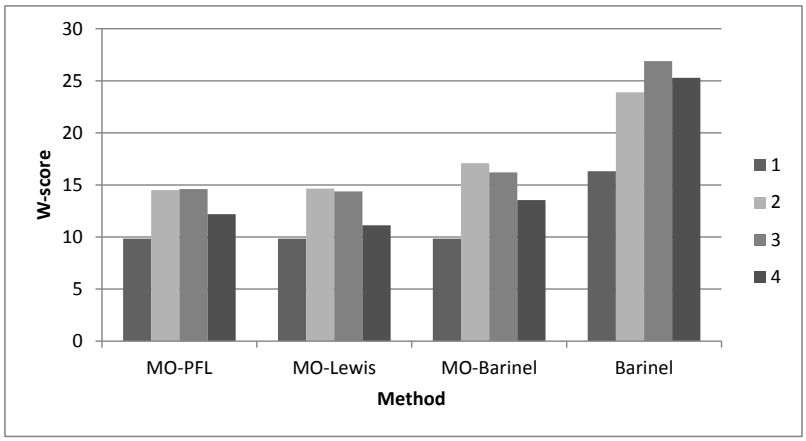

Figure 7.7: W-scores for selected techniques on SIR benchmarks

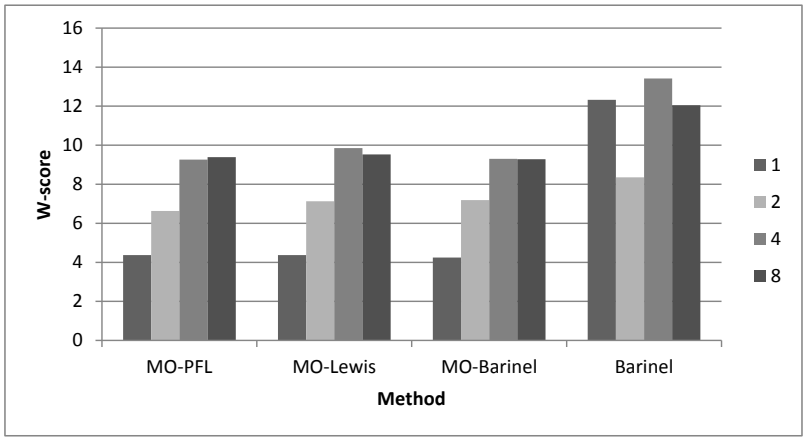

Figure 7.8: W-scores for selected techniques on Steimann benchmarks

both faults were faulty hitting sets, thus increasing the chance of the technique finding one early. Furthermore, many of the two fault programs had a small number of hitting sets ensuring the FHS was discovered on the first iteration of the use of STACCATO, meaning that faulty hitting sets consisting of two faults were found early. Several random samples of 2-fault programs from the population of 2-fault Steimann benchmarks repeated this result.

We now discuss how techniques compare in terms of efficiency in practice. We first discuss $M_{Opt(Barinel)}$'s runtime. The average runtime for the 1/2/3/4 versions of the SIR benchmarks was 1.27, 6.78, 6.29, and 7.59 seconds respectively. Thus $M_{Opt(Barinel)}$ was efficient for small programs. We now discuss the Steimann benchmarks. The av-

erage runtime for the 1/2/4/8 versions of the Steimann benchmarks was 0.24, 130.00, 43.67, and 443.08 seconds, respectively. Thus, $M_{Opt(Barinel)}$was less efficient on larger programs in general. In general, we often found using BARINEL as a value for g was either very quick, or very slow, for large individual experiments. For instance, the average runtime for the 8-fault org.htmlparser benchmarks was 1136.86 seconds (almost 20 minutes). In contrast for the 8-fault org.eclipse.draw2d benchmarks the average was 1.96 seconds. The reason for this was that the implementation of the method obtained from the authors which ranked hitting sets substantially slowed down when hitting sets became large (over 16 components or more). We emphasise that this problem might be reduced in future generations versions of the tool.

We now discuss $M_{Opt(Barinel)}$/BARINEL's timeouts. There were a small number of timeouts for the 2/4/8 versions of the Steimann benchmarks. For each of the 200 2/4/8 fault versions, there were 2/2/9 timeouts for $M_{Opt(Barinel)}$ and 2/9/31 timeouts for BARINEL. Thus the more faults there were in a program, the more likely it was for a time out. $M_{Opt(Barinel)}$ had fewer timeouts because if a component was covered by all failing traces it would would be immediately inspected due to the $Opt(g)$ component of the algorithm – which circumvents the need to call BARINEL. If a timeout was recorded, we did not penalise the score; instead the average over the successful experiments was taken. This slightly advantaged the scores of BARINEL against $M_{Opt(Barinel)}$, and thus does not effect our overall conclusion that $M_{Opt(Barinel)}$ substantially improved on BARINEL.

In contrast to BARINEL, the average run-time for $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$ was under a second. Thus using PFL and SBFL techniques were substantially more time efficient.

In summary, $M_{Opt(pfl-ppv)}$ and $M_{Opt(Lewis)}$ both substantially and statistically significantly outperform the BARINEL tool in terms of W-scores, but only slightly outperform $M_{Opt(Barinel)}$. Moreover, $M_{Opt(g)}$ substantially and statistically significantly outperforms BARINEL. The results confirm that our new algorithms $M_{Opt(pfl-ppv)}$ and $M_{Opt(Lewis)}$ are effective and efficient at fault localisation and outperform the state of the art in statistical multiple fault localisation techniques.

7.7 Summary

The major contributions of this chapter have been to the task of multiple fault localisation (MFL). We studied how to optimise existing ranking based methods (such

as SFBL, PFL, and BARINEL techniques) for the purposes of multiple fault localisation. To address this we presented an optimiser $M_{Opt(g)}$, which was formally proven to satisfy a newly identified property of multiple fault optimality. In experimentation our new algorithms $M_{Opt(pfl-ppv)}$ and $M_{Opt(Lewis)}$ were found to be effective and efficient at fault localisation and statistically significantly ($p0.01$) and substantially outperform SBFL techniques and the BARINEL tool at our multiple fault localisation tasks. In particular, using these new techniques one would expect to localise over 3 faults by investigating under 10 percent of each program in our experiments.

For future work, we would like to combine our technique with others in order to improve performance. In particular, because introducing more faults into a program can typically increase noise for fault localisation, we conjecture that using clustering methods (discussed in Chapter 2) to partition test suites into smaller test suites with a smaller number of faults has the potential to further improve the effectiveness of our $M_{Opt(g)}$ technique.

Chapter 8

Test-suite Optimisation

In the previous chapters we investigated methods which localise faults as a function of a given faulty program and test suite (called a proband). Thus far we have relied on the fact that the given test suite provided will be of sufficient quality as to facilitate a set of coverage vectors (SCV) which may be used effectively for fault localisation. However, in general there is no guarantee that a given test suite will be of the required quality, and so there remains the problem of how to improve test suites for the purposes of fault localisation, or alternatively generate them from scratch, with the goal of providing us with SCVs of sufficient quality for fault localization. Accordingly, in this chapter we treat this problem by developing an algorithm which, for a given faulty program, generates a SCV which satisfies properties exploitable by given fault localisation methods (namely, the single bug optimal methods discussed in previous chapters). The contributions of this chapter are as follows:

1. Building on Naish's property of single bug optimal measures [181], we identify a corresponding property of a single bug optimal test suite and SCV.

2. We present an algorithm which leverages model checkers to efficiently generate a SCV which satisfies this property.

3. In experimentation, we demonstrate that an implementation of the algorithm which uses the CBMC model checker, can be used to efficiently generate small test suites in practice. Moreover, we demonstrate that the resultant SCV can be used effectively in fault localisation using single bug optimal measures.

The rest of this chapter is organised as follows. In section 8.1 we identify desirable properties for a SCV to satisfy. In section 8.2 we present an algorithm which,

for a given proband, generates a SCV which satisfies the properties. In section 8.4 we present our experimental results. Finally, in Section 8.5 we summarise our contributions.

8.1 Properties

Our goal is to create an algorithm which, for a given for a given faulty program, generates a SCV which satisfies desirable formal properties which are exploitable by existing fault localization methods. Our aim is that the vectors in the generated SCV can either be combined with other vectors to create a larger SCV of improved quality, or alternatively be used as a standalone SCV for use in fault localization. In the rest of this section, we first present some notations, and then present some properties we would like the algorithm to satisfy.

We first introduce some new notation. Given a SCV $\mathbf{T}$ and its program model $\boldsymbol{PM}_{\mathbf{T}}$ (where $\mathbf{F}/\mathbf{P}$ is the set of failing/passing traces in $\mathbf{T}$ respectively), we identify three mutually exclusive sets:

1. $AN_{\mathbf{T}} = \{i | C_i \in \boldsymbol{PM}_{\mathbf{T}} \wedge |C_i \cap E| = |\mathbf{F}| \wedge |C_i \cap \overline{E}| = 0\}$ – specifying the set of components covered by all failing traces but no passing traces in $\mathbf{T}$.

2. $AS_{\mathbf{T}} = \{i | C_i \in \boldsymbol{PM}_{\mathbf{T}} \wedge |C_i \cap E| = |\mathbf{F}| \wedge |\mathbf{P}| > |C_i \cap \overline{E}| > 0\}$ – specifying the set of components covered by all failing traces and some (but not all) passing traces in $\mathbf{T}$.

3. $AA_{\mathbf{T}} = \{i | C_i \in \boldsymbol{PM}_{\mathbf{T}} \wedge |C_i \cap E| = |\mathbf{F}| \wedge |C_i \cap \overline{E}| = |\mathbf{P}|\}$ – specifying the set of components covered by all failing and passing traces in $\mathbf{T}$.

Each set contains indices which which we say *specifies* components in the given program model (and which also indicate the UUTs in the faulty program). e.g. index i specifies the ith element of the given program model and the ith UUT. For clarity, $AN_{\mathbf{T}}, AS_{\mathbf{T}}, AA_{\mathbf{T}}$ are labelled such that they correspond to "**A**ll failing/**N**o passing"/"**A**ll failing **S**ome passing"/"**A**ll failing **A**ll passing" respectively. Accordingly, we can describe the set of UUTs covered by all failing traces in $\mathbf{T}$ to be specified by $A_{\mathbf{T}} = AN_{\mathbf{T}} \cup AS_{\mathbf{T}} \cup AA_{\mathbf{T}}$.

Some further terminology is as follows. Assuming a given faulty program with n identified UUTs, the population test suite is the set of all possible test cases for the program, and a sample test suite is a test suite which is a subset of the population.

We denote the population SCV corresponding to the population test suite for that program as $^*\mathbf{T}$ (and $^*\mathbf{F}$ and $^*\mathbf{P}$ as the set of failing and passing traces in $^*\mathbf{T}$ respectively). A sample SCV is a SCV corresponding to a sample test suite. The unique program model for the population test suite is denoted $^*\boldsymbol{PM}$ (aka $^*\boldsymbol{PM}_{^*\mathbf{T}}$). By convention, each component in the population program model is also superscipted with a *. Finally, $\langle {}^*\boldsymbol{PM}, {}^*\mathbf{T}\rangle$ is called the population proband model. For a given sample proband model $\langle \boldsymbol{PM}, \mathbf{T}\rangle$ and its population proband model $\langle {}^*\boldsymbol{PM}, {}^*\mathbf{T}\rangle$, it is observed that $\mathbf{T} \subseteq {}^*\mathbf{T}$, and for each $i \in \mathbb{N}$ if $C_i \in \boldsymbol{PM}$ and ${}^*C_i \in {}^*\boldsymbol{PM}$, then $C_i \subseteq {}^*C_i$.

We now identify desirable properties we would like a given SCV to satisfy. In our development, we assume a single bug optimal measure w is being used, and that there is a single bug in a given faulty program. Given this, a property of a single bug optimal SCV for a given faulty program is then presented as follows.

Definition 8.1.1. A Property of a Single Bug Optimality. For a given proband and a sample proband model $\langle \boldsymbol{PM}, \mathbf{T}\rangle$, $A_{\mathbf{T}} = A_{^*\mathbf{T}}$

If a single bug optimal measure is being used, and there is a single bug in the given program, then we argue that a SCV $\mathbf{T}$ which satisfies the above property optimises fault localisation. By optimise, we mean improves (or at least does not negatively affect) fault localisation effectiveness. To see why the property is indeed a property of an optimal SCV, we observe that if a sample $\mathbf{T}$ satisfies the property, then w will rank components specified by $A_{^*\mathbf{T}}$ (which includes the fault) as either more suspicious, or as equally suspicious, than when a SCV which does not have this property is used. This is because a SCV $\mathbf{T}$ without this property will potentially rank components specified by $A_{\mathbf{T}} - A_{^*\mathbf{T}}$ as more suspicious than the fault. Note that the reason for why the above property is a property of a single bug optimal SCV, is similar to the reasoning underlying Naish's single bug optimal measures [181]. The difference is that we apply the reasoning to SCVs instead of measures. Finally, if a SCV satisfies the property, then its associated test suite is also said to satisfy a property of single bug optimality.

We now argue it is desirable that the following (stronger) property is satisfied.

Definition 8.1.2. Strong Single Fault Localisation Property. For a given proband and a sample proband model $\langle \boldsymbol{PM}, \mathbf{T}\rangle$, $AN_{\mathbf{T}} = AN_{^*\mathbf{T}}$, $AS_{\mathbf{T}} = AS_{^*\mathbf{T}}$, $AA_{\mathbf{T}} = AA_{^*\mathbf{T}}$

If this strong property is satisfied by a SCV then so is the first property of single bug optimality. We argue that a SCV which satisfies this stronger property is desirable for use with single bug optimal measures. To see this, we first observe that for a single bug optimal measure w used with the population SCV $\mathbf{*T}$, components specified in $AN_{\mathbf{*T}}$ are more suspicious than components specified in $AS_{\mathbf{*T}}$, which are in turn more suspicious than components specified in $AA_{\mathbf{*T}}$ – where components specified by the union of the three will be more suspicious than components that are not. Accordingly, generating a SCV with the strong property means that we can use w to rank three different sets of components in terms of suspiciousness, potentially advantaging the fault localisation process for any SCV for the given faulty program it is combined with. Finally, as has been demonstrated, adding passing traces to SCVs has been shown to improve fault localisation effectiveness in general [181], and thus a SCV which satisfies the stronger property is a potential improvement over one which only satisfies the (weaker) single bug optimal property, and can be used to guide the generation of small sized SCVs from scratch. Finally, if a SCV satisfies the stronger property, then its associated test suite is also said to satisfy a strong single fault localization property.

8.2 Algorithm

In this section we present an algorithm which generates a SCV for a given faulty program, where this SCV is then shown to satisfy the strong property of Definition 8.1.

We first present the assumptions, terminology, and notations underlying the algorithm. In our development we shall assume the given faulty program has only one assertion statement in it. We shall also assume that there is a UUT in the program which is always executed (in practice, this can be a UUT identified with the execution of the main function).

We now introduce some notation. Let X be a set of coverage vectors, then $Cov(X) = \{0 < i < |\boldsymbol{PM}_X||(\forall t_k \in X)c_i^k = 1\}$. Intuitively, $Cov(X)$ is the set of indices to components in $\boldsymbol{PM}_X$ which are covered by every coverage vector in X. For example, for the failing coverage vectors $\mathbf{F}$ represented in Table 3.1 we have $Cov(\mathbf{F}) = \{3\}$. In addition, a set of indices to components is also said to *specify* the UUTs it corresponds to. Accordingly the number specified in the latter set also specifies the UUT labelled `C3` in Figure 3.1.

We shall also make use of the following subroutine which models the high-level capabilities of existing model checkers, like CBMC [1]. $mc(b, pr, X, Y)$ is described as follows, where b is a Boolean variable, pr is a string which is the name of the faulty program, and X and Y are sets of natural numbers which specify UUTs. The subroutine functions as follows. If $b = 1$, and there is some test case which violates the specification for the program pr (and the provisos given by X and Y are met), then mc returns $\{t\}$, where t is a coverage vector which corresponds to a given test case. The provisos given by X and Y are as follows – the given test case does not execute every UUT specified in X, and executes every UUT specified in Y. A proviso is trivially fulfilled if its corresponding set is empty. If no test case exists, mc returns $\varnothing$. Finally, if $b = 0$ the process is the same except the test case in question satisfies, as opposed to violates, the specification, and t is passing coverage vector.

Following this, an algorithm which returns a SCV satisfying the strong property of single fault localization is presented in the pseudo-code of Algorithm 3. A precondition of the algorithm is that F/P are (possibly empty) sets of failing/passing traces for the faulty program named by pr. An intuition for the algorithm is as follows. The first loop generates a SCV F with the property that a UUT is executed by all failing test cases corresponding to F, if and only if it is executed by all failing test cases in the population test suite. The second loop is analogous, but holds for passing test cases P. The last loop can be thought of as generating passing coverage vectors which themselves cover a component which are firstly covered by all coverage vectors in F, and secondly not covered by all passing traces in P. The SVC returned is the union of all generated coverage vectors, and constitutes the optimal SVC.

Algorithm 3 Single-bug optimial SCV generation method

Input: name of faulty program pr, a set of failing traces F, a set of passing traces P
Output: A single bug optimal SCV T

1: $f \leftarrow mc(1, pr, Cov(F), \varnothing)$
2: $F \leftarrow F \cup f$
3: **while** $f! = \varnothing$ **do**
4: $\quad f = mc(1, pr, Cov(F), \varnothing)$
5: $\quad F \leftarrow F \cup f$
6: **end while**
7: $p \leftarrow mc(0, pr, Cov(P), \varnothing)$
8: $P \leftarrow P \cup p$
9: **while** $p! = \varnothing$ **do**
10: $\quad p = mc(0, pr, Cov(P), \varnothing)$
11: $\quad P \leftarrow P \cup p$
12: **end while**
13: $S \leftarrow \varnothing$
14: **for** $i \in Cov(F) - Cov(P)$ **do**
15: $\quad S \leftarrow S \cup mc(0, pr, \varnothing, \{i\})$
16: **end for**
17: $T \leftarrow F \cup P \cup S$
18: **return** T

Proposition 6. The set of coverage vectors returned by Algorithm 3 satisfies the Strong Single Fault Localisation Property.

Proof. Let $\langle \boldsymbol{PM}, \mathbf{T} \rangle$ be a sample proband model for a given proband, and $\langle \boldsymbol{*PM}, \mathbf{*T} \rangle$ be the population proband model. We must show $AN_{\mathbf{T}} = AN_{\mathbf{*T}}$, $AS_{\mathbf{T}} = AS_{\mathbf{*T}}$, and $AA_{\mathbf{T}} = AA_{\mathbf{*T}}$. To show these, it is sufficient to show that each of the following three conditions hold. Where $C_i, E \in \boldsymbol{PM}$ and ${}^*C_i, {}^*E \in \boldsymbol{*PM}$, we must show: First, $|C_i \cap E| = |\mathbf{F}|$ if and only if $|{}^*C_i \cap {}^*E| = |\mathbf{*F}|$. Secondly, $|C_i \cap \overline{E}|$ $= |\mathbf{P}|$ if and only if $|{}^*C_i \cap \overline{{}^*E}| = |\mathbf{*P}|$. Third, $0 < |C_i \cap \overline{E}| < |\mathbf{P}|$ and $|C_i \cap E| = |\mathbf{F}|$ if and only if $0 < |{}^*C_i \cap \overline{{}^*E}| < |\mathbf{*P}|$ and $|{}^*C_i \cap {}^*E| = |\mathbf{*F}|$.

To show the first condition, we observe a post-condition of the first loop is that F has the property that C_i is covered by all failing vectors in F just in case $*C_i$ is covered by all failing vectors in the population $\mathbf{*T}$. Further, if F has this property then so does $\mathbf{T} = F \cup P \cup S$ (as $P \cup S$ contains passing traces). Thus the first condition holds. The second condition is similar to the first. To show the third condition, we observe that a post-condition of the third loop is that for each i in $Cov(F) - Cov(P)$, S has the property that C_i is covered by a vector in S if and only if ${}^*C_i$ is covered by

all failing and some (but not all) passing vectors in ***T** (given a precondition of the loop that $Cov(F) - Cov(P) = Cov(\mathbf{*F}) - Cov(\mathbf{*P})$). Thus the third condition holds. □

Our algorithm is similar to the model checking approaches of Gopinath [96] (discussed in detail in the literature review of Chapter 2), except our approach is coverage based, computes three different sets of increasingly suspicious components, leverages passing traces, and is designed for use with single bug optimal measures.

8.3 Example

We illustrate Algorithm 3 by showing how the SVC T returned by the algorithm is constructed, using the running example of program 3.1 – the C program `minmax.c`. We use the CBMC model-checker as the model checker leveraged by the *mc* function in our practical implementation of the algorithm. For details about CBMC see [1]. To illustrate how the algorithm can be used in conjunction with CBMC in an implementation, we first describe a pre-processing stage for `minmax.c`. Firstly for each UUT we declare a global variable `int Ci = 0;`. Secondly, we insert `Ci = 1;` in the predicate or function representing that UUT, such that that line of code will be executed just in case that predicate/function is executed. The result of this pre-processing stage is presented in Figure 8.1. Following our assumption that there is a UUT which is always executed, we have added the UUT labelled `C5` next to the assertion statement.

We now illustrate the algorithm using our running example. We assume $F = P = \varnothing$ as inputs to the algorithm, and let *pr* name the preprocessed `minmax.c` program given in Figure 8.1. We begin with the first five lines. At line 1 we call $mc(1, \texttt{minmax.c}, Cov(F), \varnothing)$. Here, $Cov(F) = \varnothing$ (because $F = \varnothing$). Using CBMC, a test case which violates the specification expressed by the assertion `assert(least <= most)` is found with corresponding coverage vector $t_1 = \langle 0, 1, 1, 0, 1, 1, 1 \rangle$. Here, the first arguments represent whether UUT `C1, ... C5` were covered by the test case, the second last argument is 1, which denotes that it is a failing trace, and the last argument is a unique identifying number for the coverage vector – as per the definition of coverage vectors in 3.

At line 2, we add this to the SCV $F = \{t_1\}$. At line 3, f is assigned $t1$. Thus we enter the loop at line 4. $mc(1, \texttt{minmax.c}, Cov(F), \varnothing)$ is called. Here, $Cov(F) = \{2, 3, 5\}$. This result is obtained as follows. Following our definition of $Cov(F) = \{0 < i < PM_F | (\forall t_k \in F) c_i^k = 1\}$, we have $t_1 = \langle 0, 1, 1, 0, 1, 1, 1 \rangle$, and as $c_2^1 = c_3^1 = c_5^1 = 1$, it follows $Cov(F) = \{2, 3, 5\}$ (note it is also true that $c_6^1 = c_7^1 = 1$,

```
int C1 = 0;
int C2 = 0;
int C3 = 0;
int C4 = 0;
int C5 = 0;

int main() {

  int input1, input2, input3;
  int least = input1;
  int most = input1;

  if (most < input2)
  {
    C2 = 1;
    most = input2;  // C1
  }

  if (most < input3)
  {
    C2 = 1;
    most = input3; // C2
  }

  if (least > input2)
  {
    C3 = 1;
    most = input2; // C3 (fault)
  }

  if (least > input3)
  {
    C4 = 1;
    least = input3; // C4
  }

  C5 = 1; // C5

  assert(least <= most); // E
  }
```

Figure 8.1: `minmax.c` preprocessed

but these are not in $Cov(F)$ because the corresponding indices 6 and 7 are not less than PM_F as per the definition of Cov). Using CBMC, a test case which violates the specification is found which does not cover all of the UUTs corresponding to $\{2, 3, 5\}$. This vector is $t_2 = \langle 0, 0, 1, 1, 1, 1, 2\rangle$. (Note that in an implementation, this test case can be found by replacing the assertion with `if(!(least <= most) && !(C2 && C3`

`&& C5)) assert(0);` and calling CBMC). Thus at line 5, $F = \{t_1, t_2\}$.

We then proceed to line 3 for a third time. $f = t_2$ and thus we proceed to line 4. Comparing the two vectors in F, we have $Cov(F) = \{3, 5\}$ (note that 2 not longer features, because $c_2^2 = 0$). Thus $mc(1, minmax.c, \{3, 5\}, \varnothing)$ is called, which will return a failing vector the test case of which does not execute at least one of the UUTs corresponding to `C3` or `C5` – if such a test case exists. None exist (intuitively, this is because the statement at `C5` is executed by all possible test cases, and `C3` denotes the single bug in the program meaning it will always be executed in a failing test case. In the implementation, we can verify the fact that there exists no failing trace which does not execute both of these statements by replacing the assertion with `if(!(least <= most) && !(C3 && C5)) assert(0);`, and then calling CBMC). Thus, the empty set is assigned to f at line 4, and the condition at line 3 is false. Here, $F = \{\langle 0, 1, 1, 0, 1, 1, 1\rangle, \langle 0, 0, 1, 1, 1, 1, 2\rangle\}$, where $C(F) = \{3, 5\}$. By exiting the first loop, we have thus demonstrated that $C(F)$ specifies the UUTs covered by all failing traces in the population test suite.

Flow of control now enters line 7. The process is analogous to that previously described, except in order to generate passing traces instead of failing we call mc to generate traces which satisfy the specification (accordingly setting the first argument of mc to 0). We found that the SCV generated was $P = \{\langle 1, 1, 0, 0, 1, 0, 3\rangle, \langle 0, 0, 0, 0, 1, 0, 4\rangle\}$, where $C(P) = \{5\}$. The second loop thus demonstrates that $C(P)$ specifies the UUTs executed by all passing traces in the population test suite – `C5`. Flow of control now enters the third loop, which generates a SCV S. For each UUT specified in $C(F) - C(P)$ (equivalent to $\{3\}$), mc returns generates a test case which covers it (if one exists). The test suite generated is then $S = \{\langle 0, 0, 1, 1, 1, 0, 5\rangle\}$.

Finally, T is assigned the union of F, P and S at line 17, which is $\mathbf{T} = \{\langle 0, 1, 1, 0, 1, 1, 1\rangle, \langle 0, 0, 1, 1, 1, 1, 2\rangle, \langle 1, 1, 0, 0, 1, 0, 3\rangle, \langle 0, 0, 0, 0, 1, 0, 4\rangle, \langle 0, 0, 1, 1, 1, 0, 5\rangle\}$. This SCV is represented by the coverage matrix of Table 8.1.

We now describe how the SCV T can be used in SBFL using a single bug optimal measure. First, we can construct a proband model $\langle \mathbf{PM}_T, T\rangle$ for use in SBFL. We observe that if the user uses a single bug optimal measure with this proband model, the UUT corresponding to C_3 will be inspected first (this is because only C_3 and C_5 are covered by all failing traces, and C_3 is covered by fewer passing traces than C_5, meaning that any single bug optimal measure will make the former the most suspicious component). Consequently, our example illustrates that using single bug

	C_1	C_2	C_3	C_4	C_5	E
t_1	0	1	1	0	1	1
t_2	0	0	1	1	1	1
t_3	1	1	0	0	1	0
t_4	0	0	0	0	1	0
t_5	0	0	1	1	1	0

Table 8.1: Coverage matrix for example

optimal SCVs and measures in conjunction with single bug programs can potentially be effective.

8.4 Experimentation

In this section we present results of our experiment. The task of the experiment was to firstly test if Algorithm 3 could be used to efficiently generate SCVs for small faulty programs, and secondly examine whether they could be used in conjunction with a single bug optimal measure for potentially effective fault localization (we use the Naish measure as our single bug optimal measure). We first describe our experimental setup, and then present our results.

8.4.1 Setup

We first discuss the benchmarks used. In order to perform an unbiased and high quality experiment, benchmarks needed to satisfy the following three properties:

1. Each benchmark was created by an independent source. This is desirable in order to prevent any implicit bias caused by creating benchmarks ourselves.

2. Each benchmark has an explicit formally stated specification which can be tested by a bounded model checker such as CBMC.

3. The faulty code was clearly and explicitly identified by the corresponding authors.

Unfortunately, benchmarks satisfying the above three conditions were found to be extremely rare. In general, benchmarks exist in verification research to satisfy either

#	Benchmark	LOC
1	alternating_list_false-unreach-call.c	63
2	kundu2_BUG.cil.c	628
3	list_flag_false-unreach-call.c	57
4	merge_sort_BUG.c	151
5	mutex_lock_struct.c-unsafe.cil.c	82
6	rule57_ebda_blast.c-unsafe.cil.c	280
7	rule60_list2.c-unsafe_1.cil.c	203
8	sll_to_dll_rev_BUG.c	141
9	tcas.c	173
10	test_locks_14.BUG.c	224

Table 8.2: Benchmarks

the second or third criterion, but rarely both. For instance, the available SIR benchmarks satisfy the third criterion, but not the second [7]. Furthermore, the software verification competition (SV-COMP) benchmarks satisfied the second criterion, but almost never satisfied the third [8]. Finally, it is often difficult obtaining benchmarks from authors even when usable benchmarks do in fact exist. Available benchmarks are described in Table 8.2, where we give the benchmark name, and lines of code (LOC). The modified versions of `tcas` were made available by Alex Groce via personal correspondence and were used with the EXPLAIN tool in [104]. The remaining benchmarks were identified in the repositories of SV-COMP-2014, and are available here [8].

8.4.2 Results

Our results are given in Table 8.3, which we describe as follows: **b** is the number of blocks in the program, $|\mathbf{F}|$ and $|\mathbf{P}|$ give the number of failing and passing test cases respectively in the generated test suite, the AVG row gives the overall mean of the values in the columns, the AVG $|\mathbf{F}| > 1$ row gives the overall mean for benchmarks when $|\mathbf{F}| > 1$. The first set of **A**, **B**, **W** columns give the average, best, and worst case W-scores for the Naish measure (as defined in Chapter 3). We observed that the results are the same for any single bug optimal measure, as all faults contain a single bug. The exception was test (2), which contained two faults, the W-score for which is given when Naish is used until a fault not covering all failing traces is found, from which point $M_{Opt(Naish)}$ is used. The last three columns give the **A**, **B**, and **W** scores for any SBFL or PFL measure when there is only one failing trace. This failing trace was generated by the CBMC `--beautify` method which attempts to find a

#	Benchmark	b	|F|	|P|	A	B	W	A	B	W
1	`alternating`	11	1	2	**0.00**	0.00	0.00	27.27	0.00	54.55
2	`kundu2`	117	4	7	**11.11**	0.00	22.22	36.32	0.00	72.65
3	`list`	9	1	3	**16.67**	11.11	22.22	**16.67**	0.00	33.33
4	`merge`	24	2	1	**12.50**	0.00	25.00	16.67	0.00	33.33
5	`mutex`	7	1	1	28.57	14.29	42.86	**21.42**	0.00	42.86
6	`rule57`	24	2	3	**12.50**	4.16	20.83	18.75	0.00	37.50
7	`rule60`	17	1	3	**20.59**	0.00	41.78	35.29	0.00	70.58
8	`sll`	20	5	1	**17.50**	0.00	35.00	**17.50**	0.00	35.00
9a	`tcas_v1`	15	1	2	**13.33**	0.00	16.67	40.00	0.00	80.00
9b	`tcas_v2`	15	1	5	60.00	40.00	80.00	**40.00**	0.00	80.00
9c	`tcas_v4`	15	1	2	**3.33**	0.00	6.67	40.00	0.00	80.00
9d	`tcas_v32c`	15	1	2	43.33	13.33	73.33	**36.67**	0.00	73.33
9e	`tcas_v41`	15	1	2	**3.33**	0.00	6.67	40.00	0.00	80.00
10a	`test (1)`	75	15	1	**5.33**	4.00	6.67	12.00	0.00	24.00
10b	`test (2)`	75	13	1	**8.67**	6.67	10.67	20.00	0.00	40.00
-	AVG	30.27	3.33	2.33	**17.12**	6.24	27.37	27.90	0.00	55.81
-	AVG $\|\mathbf{F}\| > 1$	22.33	2.73	0.87	**4.51**	0.99	8.07	8.08	0	16.17

Table 8.3: Comparitive W-scores at single fault localisation task

short trace. Scores with the best average W-score are given in bold. Each of the traces generated by CBMC were generated in less than a second, and therefore did not represent a challenge for the model checker, establishing the efficiency of Algorithm 3 when used in conjunction with CBMC on small programs.

We now discuss the effectiveness of using a single bug optimal measure on the generated test suites. We observe that the user would be expected to examine almost half as much code than if the user inspected all the elements of a single (short) failing trace (measured in terms of AVG W-scores, our approach scored 17.12 overall, 4.51 for test suites with more than one failing trace). The results suggest that if one is using CBMC as the underlying model checker, then generating a small handful of additional traces as per Algorithm 3 has the potential to improve fault localisation in an efficient time scale.

Finally, we observe that many of the benchmarks did not have failing traces with different coverage properties. This meant the test suites usually contained only one failing trace. Consequently fault localisation effectiveness was limited. In general, fault localisation effectiveness of coverage-based techniques is expected to increase when more failing traces are available. Thus we think that the technique's effectiveness would be improved on benchmarks which represent this feature.

8.5 Summary

In this chapter we have outlined how we can leverage model checkers (such as CBMC) to efficiently optimise test suites to satisfy a formal property of single bug optimality. Experimental results demonstrated that small optimised test suites could be generated efficiently, and that fault localisation could be performed effectively on these test suites (in particular, when more than one failing trace was available). Our findings suggest that lightweight fault localisation methods can be used in conjunction with lightweight test suite generation methods to provide fault localisation information on small programs in seconds. Finally, a prominent element of future work is to find higher quality benchmarks satisfying the desiderata of Section 8.4.1.

Chapter 9

Conclusions

In this chapter we summarise the contributions of this thesis and discuss future work.

9.1 Thesis Summary

This dissertation has been presented to corroborate the following thesis statement: That the new algorithms presented in this thesis are effective and efficient at software fault localisation and experimentally outperform state of the art statistical techniques at a range of fault localisation tasks.

To establish *effectiveness*, we showed there was statistically significantly (using p = 0.01) superior performance in terms of W-scores of our new techniques in the following respects. $m9185$ (along with the pre-established measure Kulkzynski2) significantly outperformed all other SBFL measures at the single fault localisation task in Chapter 5. PFL-PPV significantly outperformed all SBFL techniques on our large benchmarks at our single fault localisation task in Chapter 6. $M_{Opt(pfl-ppv)}$ significantly outperformed all SBFL techniques and BARINEL at multiple fault localisation tasks in Chapter 7. Our techniques' effectiveness is grounded theoretically in newly identified formal properties (such as the *fault localisation properties* and *multiple fault optimality*) – properties which existing statistical fault localisation methods (like SBFL) do not satisfy. Finally, to establish *efficiency*, we demonstrated that our new PFL and $M_{Opt(pfl-ppv)}$ techniques took negligible runtime in practice - less than a couple of seconds on average for each fault localisation problem.

We now summarise the contributions of this thesis in greater detail.

1. In Chapter 5 we performed an investigation into the existing framework of spectrum based fault localisation (SBFL). To improve on the effectiveness of SBFL techniques, we provided the following contributions:

    - We motivated and introduced many new statistical measures which could be used within this framework. New measures included large classes of causal, confirmation, predictive, and similarity measures (almost 100 in total), and many automatically generated measures.
    - We provided a large number ranking equivalence proofs to determine a large class of inequivalent measures for experimental comparison.
    - We performed a large-scale experimental comparison to determine the best performing measures. Many of the new measures performed competitively with the existing measures in experimentation – in particular one of our new measures $m9185$ outperformed all existing measures in terms of AVG A and W-scores (4.39 and 47.39 respectively), and along with Kulkzynski2, significantly outperformed all pre-established measures (using $p = 0.01$).

2. Having investigated SBFL, our second contribution was to develop an alternative framework to SBFL in Chapter 6. Our contributions were as follows:

    - We motivated, introduced, and formally developed a new formal framework which we called probabilistic fault localisation (PFL). PFL is similar to SBFL insofar as it can leverage any suspiciousness measure, and is designed to estimate the probability that a given program artefact is faulty.
    - We formally proved that the set of PFL-equations is theoretically superior to SBFL insofar as it satisfies and exploits a number of newly identified formal properties which SBFL does not.
    - We formally presented the PFL-algorithm, which facilitates updates of fault probability as the user's investigation of the program progresses.
    - On large scale experimentation we demonstrated that PFL-Suppes outperformed the pre-established SBFL measures at finding a fault in large fault programs in terms of AVG A and W-scores (5.12 and 53.67 respectively). In addition, PFL-PPV was shown to statistically significantly and substantially outperform SBFL measures on our large benchmarks at A and W-scores (1.88 and 76.20 respectively).

- We demonstrated that for over a quarter of our large benchmarks it is theoretically impossible to design strictly rational SBFL measures which outperform PFL-PPV.
- We demonstrated that the PFL-algorithm is time efficient for practical fault localisation tasks, taking on average 1.80 seconds per fault localisation problem in our implementation.

3. Having developed techniques for finding a *single* fault in a program, we addressed the problem of finding *multiple* faults in Chapter 7. Accordingly, our third set of contributions were as follows:

   - We motivated and introduced a new algorithm $M_{Opt(g)}$ which adapts any ranking based fault localisation method g to the task of finding a faulty hitting set.
   - We formally proved that $M_{Opt(g)}$ satisfies and exploits a newly identified property of multiple-fault optimality.
   - On large scale experimentation we demonstrated that $M_{Opt(g)}$ can be used as an effective multiple fault localisation method (using Lewis and PFL-PPV as values for g), achieving W-scores of 10.00% for finding a faulty hitting consisting of an average of 3.79 and 3.69 faults respectively. Both $M_{Opt(Lewis)}$ and $M_{Opt(pfl-ppv)}$ statistically significantly and substantially improves given SBFL and BARINEL methods at W-scores (using p = 0.01) on respective fault localisation tasks.
   - We demonstrated that the $M_{Opt(g)}$ is time efficient for practical fault localisation tasks, taking on average under a second per fault localisation problem when the underlying technique g is a PFL or SBFL technique.

4. Having developed methods for localising faults as a function of a given test suite, we finally addressed the problem of optimising/generating small test suites themselves in Chapter 8. Accordingly, our fourth set of contributions were as follows:

   - We presented an algorithm which leverages model checkers to optimise test suites in order satisfy a property of single bug optimality.

- We experimentally demonstrated that on small benchmarks single bug optimal test suites can be generated efficiently when the algorithm is used in conjunction with the CBMC model checker, and that the resulting test suite can be used effectively in fault localisation.

9.2 Future Work

In this section we discuss future work. This includes developing tools, performing usability studies, and applying the techniques in this thesis to fault localisation tasks in concurrent programs and hardware.

9.2.1 Tool development and usability studies

We would like to implement our PFL and $M_{Opt(g)}$ techniques into an easy-to-use tool appropriate for the general programmer. Despite over a decade of research, manual debugging methods still seem to be the norm: As Sahoo et al. observe [203] "to our knowledge, *no automated fault localization or bug diagnosis tools are used in real world development*" (emphasis theirs). We might speculate on the reasons for this – it might partly be because many tools are difficult to use. Any tool developed must address this issue as serious in order to solve the sorts of industrial problems outlined in Section 1.1.

To this end, after developing the tool we would like to perform a series of usability studies. The study of Parnin and Orso demonstrated that fault localisation tools (such as Tarantula) aid advanced programmers in in practical fault localisation tasks [188]. We would like to perform a further study on how well tools implementations PFL and $M_{Opt(g)}$ improve the effectiveness and efficiency of engineers newly introduced to the tool.

9.2.2 Other applications

The work in this thesis has focused on the localisation of faults in single-threaded programs. However, there are many other problem areas of fault localisation for which PFL might potentially be developed and applied. These include fault localisation in multi-threaded (concurrent) programs (here, PFL must be adapted to deal with interleaving test cases), hardware fault localisation (here, fault localisation problems can include finding faults in Verilog/VHDL programs which express a circuit design, and

can consequently be reduced in many cases to software fault localisation problems), and fault localisation in networks and complex communicating systems.

Bibliography

[1] CBMC. http://www.cprover.org/cbmc/.

[2] Drdebug. www.drdebug.org.

[3] Gcov. http://www.gcovr.com.

[4] GNU. http://www.gnu.org/software/gdb/.

[5] GNUprof. http://sourceware.org/binutils/docs/gprof/.

[6] Microsoft visual studio debugger. https://msdn.microsoft.com/en-us/library/sc65sadd.aspx.

[7] Software artefact infrastructure repository. http://sir.unl.edu/portal/index.php.

[8] Software verification competition benchmarks 2014. https://sv-comp.sosy-lab.org/2014/benchmarks.php.

[9] TPTP. http://www.eclipse.org/tptp/.

[10] Wolfram alpha. www.Wolframalpha.com.

[11] IEEE standard glossary of software engineering terminology. Technical report, 1990.

[12] Mars climate orbiter mishap investigation board phase-I report. Technical report, NASA, November 1999.

[13] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[14] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan J. C. van Gemund. Exploiting count spectra for bayesian fault localization. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 12:1–12:10, New York, NY, USA, 2010. ACM.

[15] Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 409–414, 2009.

[16] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Abstraction, Reformulation, and Approximation (SARA)*, 2009.

[17] Rui Abreu and Arjan J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010.

[18] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC*, pages 39–46, 2006.

[19] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, pages 89–98. IEEE, 2007.

[20] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. A new Bayesian approach to multiple intermittent fault diagnosis. In *IJCAI*, pages 653–658, 2009.

[21] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE*, pages 88–99, 2009.

[22] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, Jul 1998.

[23] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. pages 143–151, 1995.

[24] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26, May 1991.

[25] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[26] Z. A. Al-Khanjari, M. R. Woodward, Haider Ali Ramadhan, and N. S. Kutti. The efficiency of critical slicing in fault localization. *Software Quality Journal*, 13(2):129–153, June 2005.

[27] Mohammad Amin Alipour and Alex Groce. Extended program invariants: Applications in testing and fault localization. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA 2012, pages 7–11, New York, NY, USA, 2012. ACM.

[28] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *ASE*, pages 520–523. IEEE Computer Society, 2011.

[29] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM.

[30] George K. Baah, Andy Podgurski, and Mary J. Harrold. The Probabilistic Program Dependence Graph and its Application to Fault Diagnosis. In *International Symposium on Software Testing and Analysis*, Seattle, WA, July 2008.

[31] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[32] A. Bandyopadhyay and S. Ghosh. Proximity based weighting of test cases to improve spectrum based fault localization. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 420–423, Nov 2011.

[33] C. Baroni-Urbani and M. W. Buser. Similarity of binary data. *Syst. Zool.*, page 251259, 1976.

[34] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *ICSE*, pages 82–91. ACM, 2006.

[35] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, and T. Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, IEEE Computer Society, mar 2001.

[36] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[37] François Bourdoncle. Abstract debugging of higher-order imperative languages. *SIGPLAN Not.*, 28(6):46–55, June 1993.

[38] J. Braun-Blanquet. *Plant sociology: the study of plant communities.* 1932.

[39] J. Roger Bray and J. T. Curtis. An ordination of the upland forest communities of southern wisconsin. *Ecological Monographs*, 27(4):325–349, 1957.

[40] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with Tarantula. In *ISSRE*, pages 137–146, 2007.

[41] Kai-Yuan Cai, Tao Jing, and Cheng-Gang Bai. Partition testing with dynamic partitioning. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 2, pages 113–116 Vol. 1, July 2005.

[42] Jose Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 257–267. IEEE, 2013.

[43] R. Carnap. In *Logical Foundations of Probability*. Chicago: University of Chicago Press, 1962.

[44] Peggy Cellier, Mireille Ducass, Sbastien Ferr, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243. Knowledge Systems Institute Graduate School, 2011.

[45] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 73–82, New York, NY, USA, 2004. ACM.

[46] P. Cheng. From covariation to causation: A causal power theory. *Psychological Review*, 104:367–405, 1997.

[47] S. S. Choi, S. H. Cha, and C. Tappert. A Survey of Binary Similarity and Distance Measures. *Journal on Systemics, Cybernetics and Informatics*, 8:43–48, 2010.

[48] Edmund Clarke and Helmut Veith. *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, chapter Counterexamples Revisited: Principles, Algorithms, Applications, pages 208–224. Springer Berlin Heidelberg, 2003.

[49] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[50] L.C Cole. The measurement of interspecific association. *Ecology*, page 411424, 1949.

[51] Collofello and Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, pages 745–770, 1989.

[52] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[53] Vincenzo Crupi, Katya Tentori, and Michel Gonzalez. On bayesian measures of evidential support: Theoretical and empirical issues. *Philosophy of Science*, (74):229–252, 2007.

[54] 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.

[55] Brian C. Dean, William B. Pressly, Brian A. Malloy, and Adam A. Whitley. A linear programming approach for automated localization of multiple faults. In *ASE*, pages 640–644. IEEE Computer Society, 2009.

[56] V. Debroy and W. E. Wong. On the consensus-based application of fault localization techniques. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 506–511, July 2011.

[57] Vidroha Debroy and W. Eric Wong. Insights on fault interference for programs with multiple bugs. In *ISSRE*, pages 165–174. IEEE Computer Society, 2009.

[58] Vidroha Debroy and W. Eric Wong. On the equivalence of certain fault localization techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pages 1457–1463, 2011.

[59] Vidroha Debroy and W. Eric Wong. A consensus-based strategy to improve the quality of fault localization. *Softw., Pract. Exper.*, 43(8):989–1011, 2013.

[60] Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *J. Syst. Softw.*, 90:45–60, April 2014.

[61] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM.

[62] E. Diaz, J. Tuya, and R. Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 310–313, Oct 2003.

[63] Nicholas DiGiuseppe and James A. Jones. On the influence of multiple faults on coverage-based fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 210–220. ACM, 2011.

[64] Nicholas DiGiuseppe and James A. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering (Online first article)*, March 2014.

[65] M. Dowson. The ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84.

[66] Kroeber A. L. Driver, H. E. Quantitative expression of cultural relationships. *The University of California Publications in American Archaeology and Ethnology*, pages 211–256, 1932.

[67] J. Earman. In *Bayes or Bust? A Critical Examination of Bayesian Confirmation Theory*. MIT Press., 1992.

[68] J. C. Edwards. Method, system, and program for logging statements to monitor execution of a program. *patent US6539501*, 2003.

[69] E. Eells. Probabilistic causality. CUP, 1991.

[70] Ellery Eells and Brandon Fitelson. Symmetries and asymmetries in evidential support. *Philosophical Studies*, 107:129–142, 2002.

[71] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, February 2010.

[72] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[73] B. Everitt. *The Cambridge Dictionary of Statistics*. CUP, 2002.

[74] McGowan JA. Fager EW. Zooplankton species groups in the north pacific. *Science*, pages 453–460, 1963.

[75] Daniel P. Faith. Asymmetric binary similarity measures. *Oecologia*, 57(3):287–290, 1983.

[76] Robert Feldt, Simon M. Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. *CoRR*, abs/1506.03482, 2015.

[77] H. Finch. Confirming power of observations metricized for decisions among hypotheses. *Philosophy of Science*, pages 293–307, 1962.

[78] B. Fitelson and C. Hitchcock. Probabilistic measures of causal strength. In Phyllis McKay Illari, Federica Russo, and Jon Williamson, editors, *Causality in the Sciences*. Oxford University Press, Oxford, 2011.

[79] R. Fletcher and W. Suzanne. In *Clinical epidemiology: the essentials*. Lippincott Williams and Wilkins, 2005.

[80] Stephen A. Forbes. On the local distribution of certain illinois fishes: An essay in statistical ecology. *Urbana, Ill: Illinois State Laboratory of Natural History*, 1907.

[81] Telcordia Technologies (formerly Bellcore). Xsuds users manual. 1998.

[82] Jean franc Bergeretti and Bernard A. Carry. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7:37–61, 1985.

[83] Haim Gaifman and Marc Snir. Probabilities over rich languages, testing, and randomness. *Journal of Symbolic Logic*, 47:495–548, 1982.

[84] Y. Gao, Z. Zhang, L. Zhang, C. Gong, and Z. Zheng. A theoretical study: The impact of cloning failed test cases on the effectiveness of fault localization. In *2013 13th International Conference on Quality Software*, pages 288–291, July 2013.

[85] Paul Gastin, Pierre Moro, and Marc Zeitoun. *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004. Proceedings*, chapter Minimization of Counterexamples in SPIN, pages 92–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[86] D. Gillies. *Philosophical Theories of Probability*. Taylor and Francis, 2012.

[87] Cheng Gong, Zheng Zheng, Yunqian Zhang, Zhenyu Zhang, and Yunzhi Xue. Factorising the multiple fault localization problem: Adapting single-fault localizer to multi-fault programs. In *19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012*, pages 729–732, 2012.

[88] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. Diversity maximization speedup for fault localization. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 30–39, New York, NY, USA, 2012. ACM.

[89] A. Gonzalez. Automatic error detection techniques based on dynamic invariants. *M.S. thesis, Delft University of Technology, The Netherlands*, 2007.

[90] A. Gonzalez-Sanchez, E. Piel, H. G. Gross, and A. J. C. van Gemund. Prioritizing tests for software fault localization. In *2010 10th International Conference on Quality Software*, pages 42–51, July 2010.

[91] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan JC Van Gemund. An empirical study on the usage of testability information to fault localization in software. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, page 1398–1403, Tunghai University, Taichung, Taiwan, March 2011. ACM, ACM.

[92] Alberto Gonzlez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 83–92. IEEE Computer Society, 2011.

[93] I.J. Good. A causal calculus I. *British Journal for the Philosophy of Science*, 11:305–18, 1961.

[94] I.J. Good. A causal calculus II. *British Journal for the Philosophy of Science*, 12:43–51, 1962.

[95] Irving John Good. Good thinking: The foundations of probability and its applications. *Minneapolis: University of Minnesota Press*, 1983.

[96] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 40–49, New York, NY, USA, 2012. ACM.

[97] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, (1):54–72, 2012.

[98] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[99] J. C. Gower and J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 1971.

[100] J. C. Gower and P. Legendre. Metric and euclidean properties of dissimilarity coefficients. *Journal of Classification*, 3(1):5–48, 1986.

[101] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electron. Notes Theor. Comput. Sci.*, 174(4):95–111, May 2007.

[102] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault Localization Using a Model Checker. *Softw. Test. Verif. Reliab.*, 20(2):149–173, June 2010.

[103] Alex Groce. Error explanation with distance metrics. In *TACAS*, pages 108–122. Springer, 2004.

[104] Alex Groce. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, 2005.

[105] Alex Groce, Daniel Kroening, and Flavio Lerda. *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, chapter Understanding Counterexamples with explain, pages 453–456. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[106] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *In SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[107] Michael Grottke and KS Trivedi. A classification of software faults. In *Supplemental Proc. Sixteenth International Symposium on Software Reliability Engineering*, pages 4.19–4.20, 2005.

[108] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 263–272, New York, NY, USA, 2005. ACM.

[109] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, January 2002.

[110] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering*, 35(4):484–496, July 2009.

[111] Dan Hao, Lu Zhang, Hao Zhong, Hong Mei, and Jiasu Sun. Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. In *ICSM*, pages 683–686, 2005. This effort is sponsored by the National 973 Key Basic Research and Development Program No. 2002CB312003, the State 863 High-Tech Program No. 2004AA112070, and the National Science Foundation of China No. 60403015.

[112] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 267–280. Springer, 2004.

[113] John P. Heinz. *The hollow core : private interests in national policy making*. Harvard University Press Cambridge, Mass, 1993.

[114] E. Hellinger. Neue begrndung der theorie quadratischer formen von unendlichvielen vernderlichen. *Journal fr die reine und angewandte Mathematik*, 136:210–271, 1909.

[115] R. M. Hierons. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, 38(3):629–641, May 2012.

[116] Robert M. Hierons. Verdict functions in testing with a fault domain or test hypotheses. *ACM Trans. Softw. Eng. Methodol.*, 18(4):14:1–14:19, July 2009.

[117] Christopher Hitchcock. Probabilistic causation. *The Stanford Encyclopedia of Philosophy (Winter 2012 Edition)*.

[118] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 420–425, 2012.

[119] Franz Huber. Confirmation and induction. http://www.iep.utm.edu/conf-ind/.

[120] 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.

[121] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, pages 37–50, 1912.

[122] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 247–258, New York, NY, USA, 2016. ACM.

[123] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 167–178. ACM, 2008.

[124] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Effective and efficient localization of multiple faults using value replacement. In *International Conference on Software Maintenance (ICSM)*, pages 221–230. IEEE, 2009.

[125] Susmit Jha and Sanjit A. Seshia. Are there good mistakes? A theoretical analysis of CEGIS. In *3rd Workshop on Synthesis (SYNT)*, pages 84–99, July 2014.

[126] B. Jiang and W. K. Chan. On the integration of test adequacy, test case prioritization, and statistical fault localization. In *2010 10th International Conference on Quality Software*, pages 377–384, July 2010.

[127] Bo Jiang, W.K. Chan, and T.H. Tse. On practical adequate test suites for integrated test case prioritization and fault localization. *Quality Software, International Conference on*, 0:21–30, 2011.

[128] Wei Jin and Alessandro Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.

[129] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 16–26, New York, NY, USA, 2007. ACM.

[130] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.

[131] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477. ACM, 2002.

[132] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75.

[133] James Arthur Jones. Semi automatic fault localisation. *PhD Thesis, Georgia Tech*, 2008.

[134] James F. Joyce. In *The Foundations of Causal Decision Theory*. CUP, 1999.

[135] Xiaolin Ju, Shujuan Jiang, Xiang Chen, Xingya Wang, Yanmei Zhang, and Heling Cao. Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. *J. Syst. Softw.*, 90:3–17, April 2014.

[136] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.

[137] J. Kemeny. Degree of factual support. *Philosophy of Science*, pages 307–24, 1952.

[138] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1982.

[139] John Maynard Keynes. *A Treatise on Probability*. Dover Publications, 1921.

[140] Jeongho Kim, Jonghee Park, and Eunseok Lee. A new hybrid algorithm for software fault localization. In *IMCOM*, pages 50:1–50:8. ACM, 2015.

[141] Robert Konighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 91–100, Austin, TX, 2011. FMCAD Inc.

[142] Kevin B Korb. Lecture slides. http://www.kent.ac.uk/secl/philosophy/jw/2008.

[143] Kevin B Korb, Erik P. Nyberg, and Lucas Hope. A new causal power theory. In Phyllis McKay Illari, Federica Russo, and Jon Williamson, editors, *Causality in the Sciences*. OUP, 2011.

[144] B. Korel and J. Laski. Stad-a system for testing and debugging: user perspective. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 13–20, Jul 1988.

[145] Bogdan Korel. Pelas - program error-locating assistant system. *IEEE Trans. Software Eng.*, 14(9):1253–1260, 1988.

[146] H. Kyburg. Recent work in inductive logic. *American Philosophical Quarterly*, 1(4):249–287, 1964.

[147] G. N. Lance and W. T. Williams. A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. *The Computer Journal*, 9(4):373–380, February 1967.

[148] David Landsberg, Hana Chockler, and Daniel Kroening. Probabilistic fault localisation. In *Haifa Verification Conference (HVC)*, LNCS, page accepted for publication. 2016.

[149] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. Evaluation of measures for statistical fault localisation and an optimising scheme. In *FASE*, volume 9033 of *LNCS*, pages 115–129. Springer, 2015.

[150] Tien-Duy B. Le, Ferdian Thung, and David Lo. Theory and practice, do they match? a case with spectrum-based fault localization. *2013 IEEE International Conference on Software Maintenance*, 0:380–383, 2013.

[151] Hua Jie Lee, Lee Naish, and Kotagiri Ramamohanarao. Effective software bug localization using spectral frequency weighting function. In *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010, Seoul, Korea, 19-23 July 2010*, pages 218–227, 2010.

[152] Yan Lei, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *COMPSAC*, pages 1–10. IEEE Computer Society, 2012.

[153] I. Levi. Corroboration and rules of acceptance. *British Journal for the Philosophy of Science*, pages 307–13, 1962.

[154] D. Lewis. Postscripts to "causation". In *Philosophical Papers, Volume 2*. OUP, 1986.

[155] David Lewis. Causation. *The journal of philosophy*, 70(17):556–567, 1974.

[156] Feng Li, Wei Huo, Congming Chen, Lujie Zhong, Xiaobing Feng, and Zhiyuan Li. Effective fault localization based on minimum debugging frontier set. In *CGO*, page 10. IEEE Computer Society, 2013.

[157] Jenny Li, David Weiss, Eric W. Wong, and Xiao Ma. An Integrated Solution for Testing and Analyzing Java Applications in an Industrial Setting. Technical report, Avaya Labs, 2005.

[158] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, pages 15–26, 2005.

[159] J. L. Lions. Ariane 5: Flight 501 failure. *Report, European Space Agency (ESA)*, 1996.

[160] Bing Liu, Lucia Lucia, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. Localizing multiple faults in simulink models. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[161] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, 2006.

[162] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: Statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, pages 286–295, 2005.

[163] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[164] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[165] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Automated Software Engineering (ASE)*, pages 127–138. ACM, 2014.

[166] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. Are faults localizable? In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 74–77, 2012.

[167] James R. Lyle and Mark Weiser. Automatic Program Bug Location by Program Slicing. In *2nd International Conference on Computers and Applications*, pages 877–882, Peking, 1987.

[168] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling java programs for diagnosis. In *In Proceedings of the European Conference on Artificial Intelligence (ECAI*, page 2000. Press, 2000.

[169] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 115–124, New York, NY, USA, 2010. ACM.

[170] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *ASE*, pages 128–137, 2008.

[171] Wolfgang Mayer, Markus Stumptner, and Franz Wotawa. Debugging program exceptions. In *IN PROC. DX03 WORKSHOP*, pages 119–124, 2003.

[172] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, 2 edition, 2004.

[173] Scott McMaster and Atif M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *ICSM*, pages 335–344. IEEE, 2007.

[174] Peter Milne. $\log[P(h|eb)/P(h/b)]$ is the one true measure of confirmation. *Philosophy of Science*, 63(1):21–26, March 1996.

[175] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *ICSE*, pages 142–151. ACM, 2006.

[176] Varun Modi, Subhajit Roy, and Sanjeev K. Aggarwal. Exploring program phases for statistical bug localization. In *PASTE*, pages 33–40, 2013.

[177] M. D. Mountford. An index of similarity and its application to classifica- tory problems. *Progress in Soil Zoology*, pages 43–50, 1962.

[178] L. Naish, H. J. Lee, and K. Ramamohanarao. Statements versus predicates in spectral bug localization. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 375–384, Nov 2010.

[179] Lee Naish and Hua Jie Lee. Duals in spectral fault localization. In *Australian Conference on Software Engineering (ASWEC)*, pages 51–59. IEEE, 2013.

[180] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. Spectral debugging with weights and incremental ranking. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, 0:168–175, 2009.

[181] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, pages 1–11, 2011.

[182] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. Spectral debugging: How much better can we do? In *Australasian Computer Science Conference (ACSC)*, pages 99–106, 2012.

[183] Li W. H. Nei, M. Mathematical model for studying genetic variation in terms of restriction endonucleases.

[184] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[185] A. Ochiai. Zoogeographical studies on the soleoid fishes found in Japan and its neighboring regions. *Bull. Jap. Soc. sci. Fish.*, pages 526–530, 1957.

[186] Edward E. Ogheneovo. Software dysfunction: Why do software fail? *Journal of Computer and Communications*, 2:25–35, 2014.

[187] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184. ACM, 1984.

[188] Chris Parnin and Alessandro Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209, 2011.

[189] J. Pearl. Probabilities of causation: three counterfactual interpretations and their identification. *Synthese*, 1-2(121):93–149, 1999.

[190] K. Pearson. On the theory of contingency and its relation to association and normal correlation, 1904.

[191] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society.

[192] K. R. Popper. Degree of factual support. *Journal for the Philosophy of Science*, 5:143–49, 1954.

[193] K. R. Popper. In *The Logic of Scientific Discovery*. London: Hutchinson, 1959.

[194] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. *Arxiv preprint cs.SE/0310040*, 2003.

[195] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, September 8–10, 2003.

[196] Kavita Ravi and Fabio Somenzi. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, chapter Minimal Assignments for Bounded Model Checking, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[197] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[198] N. Rescher. Theory of evidence. *Philosophy of Science*, 25:83–94, 1960.

[199] Lance J. Rips. Two kinds of reasoning. *Psychological Science, American Psychological Society*, 12:129–134, 2001.

[200] Jeremias Robetaler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 309–319, New York, NY, USA, 2012. ACM.

[201] David S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 92–104. ACM, 1992.

[202] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21, 1995.

[203] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. *SIGPLAN Not.*, 48(4), March 2013.

[204] R. Santelices, J. A. Jones, Yanbing Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering*, pages 56–66, May 2009.

[205] Viktor Schuppan and Armin Biere. *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, chapter Shortest Counterexamples for Symbolic Model Checking of LTL with Past, pages 493–509. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[206] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[207] Peter HA Sneath, Robert R Sokal, et al. *Numerical taxonomy. The principles and practice of numerical classification.* 1973.

[208] Robert Sokal and Peter Henry Andrews Sneath. *Principles of Numerical Taxonomy.* W. H. Freeman, London, 1963.

[209] T. Sorgenfrei. Molluscan assemblages from the marine middle miocene of south jutland and their environments. *Danmarks Geologiske Undersgelse*, page 503, 1958.

[210] C. Spearman. The proof and measurement of association between two things. by c. spearman, 1904. *The American journal of psychology*, 100(3-4):441–471, 1987.

[211] Friedrich Steimann and Mario Bertschler. A simple coverage-based locator for multiple faults. In *ICST*, pages 366–375. IEEE Computer Society, 2009.

[212] Friedrich Steimann and Marcus Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pages 121–130, 2012.

[213] 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 *International Symposium on Software Testing and Analysis (ISSTA)*, pages 314–324. ACM, 2013.

[214] William N. Sumner and Xiangyu Zhang. Algorithms for automatically computing the causal paths of failures. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2009.

[215] William N. Sumner and Xiangyu Zhang. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010. ACM.

[216] William N. Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. *2013 35th International Conference on Software Engineering (ICSE)*, 00:272–281, 2013.

[217] P. Suppes. In *A Probabilistic Theory of Causality*. Amsterdam, 1970.

[218] A. B. Taha, S. M. Thebaut, and S. S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, 1989.

[219] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right interestingness measure for association patterns. KDD '02, pages 32–41. ACM, 2002.

[220] K. Tarwid. Estimating the convergence of ecological niches of species assessing the likelihood of their getting together in the fishery (in polish). *Ekologia Polska*, pages 115–130, 1960.

[221] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *U.S. Department of Commerce, National Institute of Standards and Technology (NIST)*, 2002.

[222] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[223] Graham Carver Paul Cheak Tomer Katzenellenbogen Tom Britton, Lisa Jeng. Reversable debugging software. *Cambridge MBA presentation*, 2013.

[224] C. J. Van Rijsbergen. Information retrieval, 1979.

[225] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[226] Lszl Vidcs, rpd Beszdes, Dvid Tengeri, Istvn Siket, and Tibor Gyimthy. Test suite reduction for fault detection and localization: a combined approach. In *Proceedings of CSMR-WCRE 2014 Software Evolution Week (European Conference on Software Maintenance and Reengineering and Working Conference on Reverse Engineering)*, pages 204–213. IEEE Computer Society, Feb 2014.

[227] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 44–53, Nov 1998.

[228] Shaowei Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau. Search-based fault localization. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 556–559, Nov 2011.

[229] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, 2005.

[230] Zheng Wei and Bai Han. Multiple-bug oriented fault localization: A parameter-based combination approach. In *Seventh International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 18-20 June 2013 - Companion Volume*, pages 125–130, 2013.

[231] Mark Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE Press, 1981.

[232] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979.

[233] W. Wen. Software fault localization based on program slicing spectrum. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1511–1514, June 2012.

[234] Frank Wilcoxon. Individual comparisons by ranking methods. In *Biometrics Bulletin*, volume 1, pages 80–83, 1945.

[235] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, (99), 2016.

[236] W. E. Wong, T. Sugeta, Yu Qi, and J. C. Maldonado. Smart debugging software architectural design in sdl. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 41–47, Nov 2003.

[237] W. Eric Wong and Vidroha Debroy. A survey of software fault localization, 2009.

[238] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani M. Thuraisingham. Effective software fault localization using an RBF neural network. *IEEE Trans. Reliability*, 61(1):149–169, 2012.

[239] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 42(3):378–396, 2012.

[240] W. Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, pages 891–903, 2006.

[241] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *COMPSAC*, pages 449–456, 2007.

[242] W.E. Wong, V. Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *Reliability, IEEE Transactions on*, 63(1):290–308, March 2014.

[243] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In Ji Wang, W. K. Chan, and Fei-Ching Kuo, editors, *QSIC*, pages 161–170. IEEE Computer Society, 2010.

[244] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *IEA/AIE*, pages 746–757, 2002.

[245] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *Developments in Applied Artificial Intelligence*, volume 2358 of *LNCS*, pages 746–757. 2002.

[246] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, pages 31:1–31:40, 2013.

[247] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. *SIGPLAN Not.*, 43(6):238–248, June 2008.

[248] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.

[249] Jifeng Xuan and Martin Monperrus. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *ICSME*, 2014.

[250] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.

[251] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell: Fault localization using time spectra. In *ICSE*, pages 81–90. ACM, 2008.

[252] Cemal Yilmaz and Clay Williams. An automated model-based debugging approach. In *ASE*, pages 174–183. ACM, 2007.

[253] S. Yoo, X. Xiaoyuan, F. Kuo, T. Chen, Y. Tsong Yueh, and M. Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *Department of Computer Science, University College London*, 2014.

[254] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *SSBSE*, volume 7515 of *LNCS*, pages 244–258, 2012.

[255] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19, 2013.

[256] Zunwen You, Zengchang Qin, and Zheng Zheng. Statistical fault localization using execution sequence. In *2012 International Conference on Machine Learning and Cybernetics*, volume 3, pages 899–905, July 2012.

[257] 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.

[258] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[259] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281. ACM, 2006.

[260] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.

[261] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. Debugging through evaluation sequences: A controlled experimental study. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 128–135, July 2008.

[262] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. Capturing propagation of infected program states. In *ESEC/FSE*, pages 43–52. ACM, 2009.

[263] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In William W. Cohen and Andrew Moore, editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM, 2006.

[264] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, pages 1105–1112, 2006.

[265] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, 2009.

Appendix A

Appendix

A.1 Established SBFL Measures

In this section of the appendix we present tables of definitions of large classes of established measures discussed in the literature review of Chapter 2. The list is almost exhaustive, save for the Dstar and combination measures discussed in Chapter 2. We present the 32 similarity measures selected by Naish [181] (Tables A.1 and A.2), the 40 association measures of Lucia [163] (Tables A.3 and A.4), and the 30 genetically learned measures of Yoo [254] (Table A.5). The first column gives the name of the measure, and the second gives the measure.

eq	Name	Measure
S1	Naish-I	-1 if $a_{ef} < F$ $P - a_{ep}$ if $a_{ef} = F$
	Naish-II	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
S2	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$
	Sorensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$
	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
	Kulczynski-I	$\frac{a_{ef}}{a_{nf}+a_{ep}}$
S3	Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$
	qe	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
	CBI Inc.	$\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{a_{ef}+a_{nf}}{a_{ef}+a_{ep}+a_{nf}+a_{np}}$
	Naish-III	$\frac{a_{ef}}{a_{ep}}$
S4	Wong-II	$a_{ef} - a_{ep}$
	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
	SimpleMatching	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{np}+a_{ep}}$
	Sokal	$\frac{2(a_{ef}+a_{np})}{2a_{ef}+2a_{nf}+a_{nf}+a_{ep}}$
	Rogers&Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
	Hamming	$a_{ef}+a_{np}$
	Euclid	$\sqrt{a_{ef}+a_{np}}$
	M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$
S5	Wong-I	a_{ef}
	Russel&Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
	Binary	0 if $a_{ef} < F$ 1 if $a_{ef} = F$
S6	Scott	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$
	Rogot-I	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$

Table A.1: Naish's measures 1/2

Name	Measure
Kulczynski-II	$\frac{1}{2}(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}})$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$
M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
Ample-I	$\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\rvert$
Ample-II	$\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}$
Wong-III	$a_{ef}-h$, where $h=a_{ep}$ if $a_{ep}\leq 2$ $h=2+0.1(a_{ep}-2)$ if $2<a_{ep}\leq 10$ $h=2.8+0.01(a_{ep}-10)$ if $a_{ep}>10$
Wong-III'	-1000 if $aep+aef=0$ Wong-III otherwise
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$
Harmonic Mean $\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np}))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$	
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$
CBI Sqrt	$\frac{2}{\frac{1}{CBI\,Inc}+\frac{\sqrt{a_{ef}+a_{nf}}}{\sqrt{a_{ef}}}}$
CBI Log	$\frac{2}{\frac{1}{CBI\,Inc}+\frac{log(a_{ef}+a_{nf})}{log(a_{ef})}}$
Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Rogot-II $\frac{1}{4}(\frac{aef}{aef+aep}+\frac{aef}{aef+anf}+\frac{anp}{aep+anp}+\frac{anp}{anf+anp})$	
F1	$\frac{2}{\frac{1}{\frac{aef}{aef+acp}}+\frac{1}{\frac{aef}{aef+anf}}}$

Table A.2: Naish's measures 2/2

Name	Measure
ϕ-coefficient	$\frac{P(E \cap C) - P(C)P(E)}{\sqrt{P(C)P(E)(1-P(C))(1-P(E))}}$
OddsRatio	$\frac{P(C \cap E)P(\overline{C} \cap \overline{E})}{P(C \cap \overline{E})P(\overline{C} \cap E)}$
Yule-Q	$\frac{P(C \cap E)P(\overline{C} \cap \overline{E}) - P(C, \overline{E})P(\overline{C} \cap E)}{P(CE)P(\overline{C} \cap \overline{E}) + P(C \cap \overline{E})P(\overline{C} \cap E)}$
Yule-Y	$\frac{\sqrt{P(C \cap E)P(\overline{C} \cap \overline{E})} - \sqrt{P(C \cap \overline{E})P(\overline{C} \cap E)}}{\sqrt{P(CE)P(\overline{C} \cap \overline{E})} + \sqrt{P(C \cap \overline{E})P(\overline{C} \cap E)}}$
Kappa	$\frac{P(C \cap E) + P(\overline{C} \cap \overline{E}) - P(C)P(E) - P(\overline{C})P(\overline{E})}{1 - P(C)P(E) - P(\overline{C})P(\overline{E})}$
J-Measure	$\max(P(C \cap E)\log(\frac{P(E\|C)}{P(E)}) + P(C \cap \overline{E})\log(\frac{P(\overline{E}\|C)}{P(\overline{E})}),$ $P(C \cap E)\log(\frac{P(C\|E)}{P(C)}) + P(\overline{C} \cap E)\log(\frac{P(\overline{C}\|E)}{P(\overline{C})}))$
Gini Index	$\max(P(C)(P(E\|C)^2 + P(\overline{E}\|C)^2) +$ $P(\overline{C})(P(E\|\overline{C})^2 + P(\overline{E}\|\overline{C})^2) - P(E)^2 - P(\overline{E})^2,$ $P(E)(P(C\|E)^2 + P(\overline{C}\|E)^2) + P(\overline{E})(P(C\|\overline{E})^2$ $+P(\overline{C}\|\overline{E})^2 - P(C)^2 - P(\overline{C})^2)$
Support	$P(C \cap E)$
Confidence	$\max((P(E\|C), P(C\|E))$
Laplace	$\max(\frac{TP(C \cap E)+1}{TP(C)+2}, \frac{TP(C \cap E)+1}{TP(E)+2})$
Conviction	$\max(\frac{P(C)P(\overline{E})}{P(C \cap \overline{E})}, \frac{P(\overline{C})P(E)}{P(\overline{C} \cap E)})$
Interest	$\frac{P(C \cap E)}{P(C)P(E)}$
Piatetsky-Shapiro	$P(C \cap E) - P(C)P(E)$
Certainty	$\max(\frac{P(E\|C) - P(E)}{P(\overline{E})}, \frac{P(C\|E) - P(C)}{P(\overline{C})})$
Added-Value	$\max(P(E\|C) - P(E), P(C\|E) - P(C))$
Coll-Strength	$\frac{P(C \cap E) + P(\overline{C} \cap \overline{E})}{P(C)P(E) + P(\overline{C})P(\overline{E})} \times$ $\frac{1 - P(C)P(E) - P(\overline{C})P(\overline{E})}{1 - P(C \cap E)) + P(\overline{C} \cap \overline{E})}$
Jaccard	$\frac{P(C,E)}{P(C) + P(E) - P(C,E)}$
Klosgen	$\sqrt{P(C \cap E)} \cdot$ Added-Value
InformationGain	$(-P(E)\log P(E) - P(\overline{E})\log P(\overline{E})) -$ $(P(C)(-P(E\|C)\log P(E\|C)) - P(\overline{E}\|C)\log P(\overline{E}\|C) -,$ $P(\overline{C})(-P(E\|\overline{C})\log P(E\|\overline{C})) - P(\overline{E}\|\overline{C})\log P(\overline{E}\|\overline{C})))$

Table A.3: Lucia's Association measures 1/2

Name	Measure
Coverage	P(C)
Accuracy	$P(C \cap E) + P(\overline{C} \cap \overline{E})$
Leverage	$P(E\|C) - P(C)P(E)$
Relative Risk	$P(E\|C)/P(E\|\neg\|C)$
Interestingness weighted	$(\frac{P(C \cap E)}{P(E)P(C)})^k) - 1)P(C \cap E)^m$
Goodman&Kruskal	$\frac{\sigma - \sigma'}{2T - \sigma'}$, where
$\sigma = \max(a_{ef}, a_{ep}) + \max(a_{nf}, a_{np})$ $+ \max(a_{ef}, a_{nf}) + \max(a_{ep}, a_{np})$, and $\sigma' = \max(a_{ef} + a_{nf}, a_{ep} + a_{np})$ $+ \max(a_{ef} + a_{ep}, a_{nf} + a_{np})$	
Mutual Information	$P(C \cap E) \log_2 \frac{P(C \cap E)}{P(C)P(E)}$
One Way Support	$P(E\|C) \log_2 \frac{P(C \cap E)}{P(C)P(E)}$
Two Way Support	$P(E \cap C) \log_2 \frac{P(C \cap E)}{P(C)P(E)}$
Loevinger	$1 - \frac{P(C)(\overline{E})}{P(C \cap \overline{E})}$
Sebag-Schoenauer	$\frac{P(E \cap C)}{P(C \cap \overline{E})}$
Least Contradition	$\frac{P(C \cap E) - P(C \cap \overline{E})}{P(E)}$
Odd Multiplier	$\frac{P(C \cap E)P(\overline{E})}{P(E)P(C\ \overline{E})}$
Example and Count.	$1 - P(C \cap \overline{E})/P(C \cap E)$
Zhang	$\frac{P(C \cap E) - P(C)P(E)}{max((P(C \cap E)P(\overline{E}), P(E)P(C \cap \overline{E})}$
Sorensoen-Dice	$\frac{2P(C \cap E)}{2P(C \cap E) + P(\overline{C} \cap E) + P(C \cap \overline{E})}$
Anderberg	$\frac{P(C \cap E)}{P(C \cap E) + 2(P(\overline{C} \cap E) + P(C \cap \overline{E}))}$
Simple Matching	$P(C \cap E) - P(\overline{C} \cap \overline{E})$
Rogers and Tanimoto	$\frac{P(C \cap E) - P(\overline{C} \cap \overline{E})}{P(C \cap E) - P(\overline{C} \cap \overline{E}) + 2(P(\overline{C} \cap E) + P(C \cap \overline{E}))}$
Ochiai-II	$\frac{a_{ef} a_{np}}{\sqrt{((a_{ef} + a_{cp}) * (a_{cf} + a_{nf}) * (a_{ep} + a_{np}) * (a_{nf} + a_{np}))}}$

Table A.4: Lucia's Association measures 2/2

ID	Measure
GP01	$a_{ef}(a_{np} + a_{ef}(1 + \sqrt{a_{ef}}))$
GP02	$2(a_{ef} + \sqrt{a_{np}} + \sqrt{a_{ep}}$
GP03	$\sqrt{\lvert a_{ef}^2 - \sqrt{a_{ep}}\rvert}$
GP04	$\sqrt{\lvert \frac{a_{np}}{a_{ep} - a_{np}} - a_{ef}\rvert}$
GP05	$\frac{(a_{ef}+a_{np})\sqrt{a_{ef}}}{(a_{ef}+a_{ep})(a_{np}a_{nf}+\sqrt{a_{ep}})(a_{ep}+a_{np})\sqrt{\lvert a_{ep}-a_{np}\rvert}}$
GP06	$a_{ef}a_{np}$
GP07	$2a_{ef}(1 + a_{ef} + \frac{1}{2a_{np}} + (1 + \sqrt{2})\sqrt{a_{np}}$
GP08	$a_{ef}^2(2a_{ep} + 2a_{ef} + 3a_{np})$
GP09	$\frac{a_{ef}\sqrt{a_{np}}}{a_{np}+a_{np}} + a_{np} + a_{ef} + a_{ef}^3$
GP10	$\sqrt{\lvert a_{ef} - \frac{1}{a_{np}}\rvert}$
GP11	$a_{ef}^2(a_{ef}^2 + \sqrt{a_{np}})$
GP12	$\sqrt{a_{ep} + a_{ef} + a_{np} - \sqrt{a_{ep}}}$
GP13	$a_{ef}(1 + \frac{1}{2a_{ep}+a_{ef}})$
GP14	$a_{ef} + \sqrt{np}$
GP15	$a_{ef} + \sqrt{a_{nf} + \sqrt{a_{np}}}$
GP16	$\sqrt{a_{ef}^{\frac{3}{2}} + a_{np}}$
GP17	$\frac{2a_{ef}+a_{nf}}{a_{ef}-a_{np}} + \frac{a_{np}}{\sqrt{a_{ef}}} - a_{ef} - a_{ef}^2$
GP18	$a_{ef}^3 + 2a_{np}$
GP19	$a_{ef}\sqrt{\lvert a_{ep} - a_{ef} + a_{nf} - a_{np}\rvert}$
GP20	$2(a_{ef} + \frac{a_{np}}{a_{ep}+a_{np}})$
GP21	$\sqrt{a_{ef} + \sqrt{a_{ef} + a_{np}}}$
GP22	$a_{ef}^2 + a_{ef} + \sqrt{np}$
GP23	$\sqrt{a_{ef}}(a_{ef}^2 + \frac{a_{np}}{a_{ef}} + \sqrt{a_{np}} + a_{nf} + a_{np})$
GP24	$a_{ef} + \sqrt{a_{np}}$
GP25	$a_{ef}^2 + \sqrt{a_{np}} + \frac{\sqrt{ef}}{\sqrt{\lvert ep-np\rvert}} + \frac{np}{(ef-np)}$
GP26	$2a_{ef}^2 + \sqrt{a_{np}}$
GP27	$\frac{a_{np}\sqrt{a_{np}a_{nf}-a_{ef}}}{a_{ef}+a_{np}a_{nf}}$
GP28	$a_{ef}(a_{ef} + \sqrt{a_{np}} + 1)$
GP29	$a_{ef}(2a_{ef}^2 + a_{ef} + a_{np}) + \frac{(a_{ef}-a_{np})\sqrt{a_{np}a_{ef}}}{a_{ep}-a_{np}}$
GP30	$\sqrt{\lvert a_{ef} - \frac{a_{nf}-a_{np}}{a_{ef}+a_{nf}}\rvert}$

Table A.5: Yoo's genetic measures

A.2 SBFL and PFL W-scores

In this section of the appendix we present the mean W-scores for each n-fault benchmark for a range of measures. We present the unavoidable cost scores (**UC**) and Random (**Rand.**) (as these gave realistic upper/lower bound scores for the SBFL approach), $m9185$ (as it achieved the highest W-scores overall for an SBFL measure), PFL-PPV (**Pfl.**) (to illustrate our main PFL technique - we have put the W-score for PFL-PPV in bold when it outscores UC). We represent $Opt(Ochiai)$ (**Opt.**), as it was the best performing single bug optimal measure, and Ochiai (**Och.**), in order to contrast against the single bug optimised variant $Opt(Ochiai)$. We also represent Tarantula (**Tar.**) (ranking equivalent to PPV) due to its prominence in the SBFL literature.

#	Benchmark	Bugs	UC	Pfl.	m9185	Opt.	Och.	Tar.	Ran.
1	gzip	1	3.75	7.50	8.75	3.75	7.50	17.27	29.15
2		2	5.78	15.59	15.52	24.87	14.32	27.16	42.18
3		3	3.54	18.69	18.52	24.36	17.03	32.71	26.22
4		4	0.00	11.90	16.67	14.29	11.90	40.48	7.14
5	print_tokens	1	2.94	3.59	3.92	2.94	6.86	21.75	49.96
6		2	1.23	5.99	7.05	10.76	4.09	9.23	27.6
7		3	0.36	3.62	1.62	5.89	3.74	1.95	21.83
8		4	0.21	1.27	1.70	5.99	1.50	0.71	21.67
9	print_tokens2	1	3.69	15.17	4.50	3.69	19.10	24.98	32.00
10		2	0.67	4.14	3.20	12.90	5.98	8.94	39.88
11		3	0.00	0.36	1.25	8.09	0.55	2.64	18.16
12		4	0.00	0.47	0.00	6.76	0.00	0.12	16.86
13	replace	1	6.42	7.58	7.04	6.42	8.08	12.38	43.77
14		2	0.89	5.12	10.82	12.06	5.24	6.92	32.85
15		3	0.23	4.28	10.10	8.92	4.07	4.72	21.23
16		4	0.25	4.92	10.31	9.92	4.65	4.05	24.86
17	schedule	1	8.18	9.16	8.18	8.18	9.50	12.55	55.46
18		2	2.17	3.63	5.10	13.34	4.20	8.60	43.51
19		3	1.26	2.70	3.49	22.33	3.09	7.81	23.67
20		4	1.15	2.23	2.41	16.87	3.22	6.04	18.07
21	schedule2	1	31.98	43.74	32.31	31.98	44.96	57.48	32.37
22		2	17.24	38.87	21.8	21.69	41.22	52.14	35.37
23		3	12.33	36.47	14.25	13.95	39.57	51.86	21.43
24		4	10.85	36.47	11.79	11.30	40.31	54.2	18.21
25	sed	1	1.04	3.96	4.79	1.04	4.38	11.81	65.22
26		2	0.71	3.79	2.96	10.90	5.17	2.96	38.09
27		3	0.46	3.66	1.84	6.91	4.73	0.84	17.05
28		4	0.00	5.56	0.00	4.44	4.44	0.00	32.22
29	space	1	2.84	4.75	3.42	2.84	4.41	14.54	50.55
30		2	0.56	1.97	1.78	10.26	1.72	4.02	29.32
31		3	0.15	1.80	1.70	6.38	1.80	2.51	23.24
32		4	0.08	1.72	0.90	5.33	1.75	2.12	21.44
33	tcas	1	29.42	38.17	30.39	29.42	38.58	45.66	40.19
34		2	10.88	25.27	17.02	14.57	25.90	32.25	28.82
35		3	6.65	20.66	14.05	10.64	21.63	28.38	20.53
36		4	3.20	16.95	9.47	5.79	18.16	26.23	13.38
37	tot_info	1	8.22	16.58	9.09	8.22	16.58	26.58	44.52
38		2	1.57	9.72	5.01	8.73	10.81	22.57	32.72
39		3	0.88	9.90	7.59	7.46	10.63	17.91	26.11
40		4	0.42	7.79	5.51	3.87	8.18	14.62	15.01
-	AVG		4.56	11.39	8.40	10.95	11.99	17.99	30.05

Table A.6: SBFL W-scores – SIR Benchmarks

#	Benchmark	Bugs	UC	Pfl.	m9185	Opt.	Och.	Tar.	Ran.
41	daikon	1	1.60	2.69	2.17	1.97	2.12	2.46	36.00
42		2	0.66	**0.61**	1.93	4.12	2.10	1.96	31.56
43		4	0.74	**0.27**	1.79	3.55	2.31	1.65	22.97
44		8	1.12	**0.15**	1.98	3.04	2.92	1.83	15.57
45		16	1.36	**0.06**	2.37	3.82	3.82	2.09	9.27
46		32	1.64	**0.13**	2.49	4.38	4.38	2.27	5.69
47	eventbus	1	7.10	7.84	8.01	6.95	8.17	11.84	46.37
48		2	1.56	2.55	3.48	7.15	3.08	6.23	33.64
49		4	0.48	0.91	1.85	3.07	1.74	3.00	20.66
50		8	0.26	0.83	1.25	2.07	1.83	1.46	12.42
51		16	0.38	0.93	1.71	2.37	2.36	1.15	6.89
52		32	0.78	**0.38**	1.98	2.25	2.25	1.44	4.02
53	jaxen	1	2.46	5.91	3.84	2.38	6.80	14.21	49.79
54		2	0.51	2.40	2.60	7.72	2.95	7.00	34.67
55		4	0.21	1.60	1.60	3.02	1.61	3.27	23.14
56		8	0.15	2.07	1.76	2.03	2.12	3.02	15.65
57		16	0.23	2.29	1.96	2.06	2.15	2.75	11.19
58		32	0.21	1.60	1.58	1.43	1.45	2.19	8.94
59	Jester1.37b	1	7.35	9.15	9.39	6.49	9.02	12.98	38.14
60		2	2.43	3.43	6.21	8.17	5.75	7.52	28.75
61		4	0.77	1.51	3.31	3.91	3.50	4.4	18.62
62		8	0.34	0.48	1.80	2.33	2.32	2.69	11.43
63		16	0.22	**0.12**	1.24	1.46	1.46	2.36	6.50
64		32	0.09	**0.01**	1.01	1.16	1.16	1.88	3.29
65	Jexel	1	4.22	6.09	5.86	4.05	7.20	12.2	45.89
66		2	0.51	1.88	3.1	5.84	3.32	6.37	35.22
67		4	0.21	1.16	1.92	3.79	2.17	2.43	21.32
68		8	0.21	1.19	1.63	2.45	2.18	1.64	11.76
69		16	0.07	1.04	1.11	2.16	2.15	0.89	6.2 0
70		32	0.07	0.49	1.26	1.89	1.89	0.54	2.88
71	JParsec	1	2.97	3.56	3.38	2.72	3.83	6.49	45.64
72		2	0.66	0.80	1.63	2.77	1.61	2.56	33.02
73		4	0.15	0.27	0.71	0.74	0.69	1.05	19.96
74		8	0.07	0.13	0.44	0.62	0.62	0.58	11.08
75		16	0.03	0.04	0.28	0.81	0.81	0.39	6.04
76		32	0.06	**0.00**	0.35	1.11	1.11	0.56	3.05
77	org.apache.commons.codec	1	12.52	13.26	13.41	11.52	12.67	20.24	39.73
78		2	2.24	2.78	4.41	4.14	4.09	7.39	28.17
79		4	0.60	**0.58**	1.71	1.52	1.52	2.83	19.19
80		8	0.28	**0.11**	1.06	0.69	0.69	1.51	10.05
81		16	0.22	**0.01**	0.97	0.44	0.44	1.13	5.44
82		32	0.16	**0.00**	0.77	0.29	0.29	0.83	2.72
83	org.apache.commons.lang3	1	4.18	4.42	4.07	2.91	3.82	6.48	31.83
84		2	0.90	**0.51**	2.63	2.59	2.31	3.55	24.23
85		4	0.34	**0.09**	1.39	1.05	1.05	2.14	16.22
86		8	0.19	**0.01**	0.95	0.58	0.58	1.38	9.36
87		16	0.12	**0.00**	0.60	0.36	0.36	0.80	4.82
88		32	0.08	**0.00**	0.45	0.33	0.33	0.42	2.54
89	org.eclipse.draw2d	1	6.99	7.97	8.08	6.53	8.01	12.1	47.22
90		2	2.22	3.48	3.91	7.92	3.97	6.21	34.62
91		4	1.01	2.31	2.94	4.97	3.42	3.84	21.73
92		8	0.91	2.13	2.65	4.30	3.93	2.31	14.09
93		16	0.99	2.11	2.94	4.90	4.88	2.05	8.48
94		32	1.51	2.07	3.57	5.26	5.26	2.16	5.35
95	org.htmlparser	1	2.06	3.16	2.63	1.84	3.38	6.27	43.25
96		2	0.61	0.94	1.85	10.46	1.69	3.43	33.01
97		4	0.24	0.80	1.48	2.48	1.37	2.75	21.87
98		8	0.22	0.91	1.59	2.18	2.07	2.34	12.73
99		16	0.37	0.72	2.07	3.27	3.26	1.84	7.59
100		32	0.65	**0.13**	2.50	3.79	3.79	1.68	4.69
-	AVG		1.39	1.88	2.63	3.34	2.97	3.88	19.77
-	AVG overall		2.65	5.69	4.91	6.38	6.58	9.53	23.88

Table A.7: SBFL W-scores – Steimann Benchmarks

A.3 MFL W-scores

In section we present the mean W-scores for each n-fault benchmark for a range of measures. The second column gives the name of the benchmark, the third the number of bugs (**Bugs**), the fourth the number of covered bugs (**cb**). **fb** gives the number of found bugs by the technique named on the immediate right, under a technique name we give the average W-score for that n-fault benchmark. We give results for $M_{Opt(g)}$ when g is Lewis and PFL-PPV, which achieved the best overall AVG W-scores of all our techniques compared. All W-scores represent the W-score for finding a faulty hitting set. We also give the results for the Lewis measure when using the SBFL algorithm to demonstrate how much $M_{Opt(Lewis)}$ improves upon its unoptimised counterpart.

#	Benchmark	Bugs	cb	fb	$M_{Opt(pfl-ppv)}$	fb	$M_{Opt(Lewis)}$	Lewis
1	gzip	1	1	1.00	3.75	1.00	3.75	5.63
2		2	2	1.83	25.31	1.83	25.31	36.74
3		3	3	2.50	25.54	2.50	27.47	62.68
4		4	4	3.00	16.67	3.00	19.05	78.57
5	print_tokens	1	1	1.00	2.94	1.00	2.94	14.38
6		2	2	1.43	11.19	1.43	11.70	28.84
7		3	3	1.50	9.97	1.50	10.9	23.62
8		4	4	1.33	8.39	1.33	9.04	14.96
9	print_tokens2	1	1	1.00	3.69	1.00	3.69	4.77
10		2	2	1.63	13.12	1.63	13.78	23.44
11		3	3	2.10	19.13	2.05	12.92	32.85
12		4	4	2.56	28.05	2.50	14.40	38.36
13	replace	1	1	1.00	6.42	1.00	6.42	6.82
14		2	2	1.74	14.61	1.74	15.16	18.42
15		3	3	2.22	16.25	2.20	16.63	20.84
16		4	4	2.50	21.09	2.44	17.72	26.40
17	schedule	1	1	1.00	8.18	1.00	8.18	8.82
18		2	2	1.26	15.93	1.26	15.78	29.45
19		3	3	1.58	27.19	1.55	27.72	36.60
20		4	4	1.33	17.59	1.33	20.38	18.02
21	schedule2	1	1	1.00	31.98	1.00	31.98	64.81
22		2	2	1.00	21.56	1.00	21.56	56.10
23		3	3	1.00	13.76	1.00	13.76	38.29
24		4	4	1.00	11.03	1.00	11.03	34.30
25	sed	1	1	1.00	1.04	1.00	1.04	8.54
26		2	2	1.56	10.90	1.56	11.09	9.19
27		3	3	1.57	7.77	1.57	8.42	7.71
28		4	4	1.00	4.44	1.00	4.44	4.44
29	space	1	1	1.00	2.84	1.00	2.84	10.19
30		2	2	1.49	10.46	1.49	10.82	18.11
31		3	3	1.72	10.09	1.72	10.20	17.78
32		4	4	1.80	8.90	1.80	8.88	22.4
33	tcas	1	1	1.00	29.42	1.00	29.42	43.7
34		2	2	1.00	13.96	1.00	13.96	21.67
35		3	3	1.00	9.47	1.00	9.47	14.24
36		4	4	1.00	4.36	1.00	4.36	5.67
37	tot_info	1	1	1.00	8.22	1.00	8.22	11.25
38		2	2	1.12	9.20	1.12	9.03	19.58
39		3	3	1.14	7.83	1.14	7.97	17.09
40		4	4	1.02	3.87	1.02	3.94	8.17
AVG		2.50	2.50	1.42	12.90	1.42	12.63	24.09

Table A.8: MFL W-scores – SIR Benchmarks

#	Benchmark	Bugs	cb	fb	$M_{Opt(pfl-ppv)}$	fb	$M_{Opt(Lewis)}$	Lewis
41	daikon	1	1.00	1.00	1.97	1.00	1.97	2.10
42		2	1.80	1.71	4.17	1.71	4.17	6.68
43		4	3.10	2.78	7.45	2.79	5.21	9.77
44		8	5.23	4.42	10.04	4.48	6.01	17.27
45		16	8.35	6.78	12.63	6.94	8.19	23.68
46		32	13.20	10.12	13.44	10.41	9.49	29.03
47	eventbus	1	1.00	1.00	6.98	1.00	6.98	7.90
48		2	1.93	1.73	9.53	1.73	9.84	26.56
49		4	3.67	2.88	9.46	2.89	10.01	44.61
50		8	6.70	4.14	8.21	4.27	9.20	52.52
51		16	11.79	5.37	7.74	5.64	8.35	51.80
52		32	18.63	6.50	7.18	6.86	8.59	45.52
53	jaxen	1	1.00	1.00	2.38	1.00	2.38	10.29
54		2	1.91	1.63	8.22	1.63	8.49	24.60
55		4	3.51	2.42	6.61	2.44	7.23	31.76
56		8	6.09	2.90	6.21	2.92	6.13	28.53
57		16	8.99	3.28	6.55	3.28	5.52	24.23
58		32	12.11	4.05	6.71	4.00	5.20	24.02
59	Jester1.37b	1	1.00	1.00	6.47	1.00	6.47	10.03
60		2	1.94	1.81	10.22	1.81	11.09	29.79
61		4	3.65	2.95	10.41	3.06	10.98	40.22
62		8	6.50	4.22	8.09	4.67	10.53	46.43
63		16	10.88	5.54	5.43	6.64	10.40	46.42
64		32	16.86	7.84	6.26	9.52	12.64	42.36
65	Jexel	1	1.00	1.00	4.00	1.00	4.00	7.72
66		2	1.87	1.65	6.86	1.65	7.56	18.22
67		4	3.50	2.64	7.97	2.07	9.34	27.37
68		8	6.50	4.08	9.54	4.42	9.89	30.49
69		16	11.41	5.60	8.14	6.38	7.61	21.30
70		32	18.56	7.27	6.32	8.44	5.62	13.38
71	JParsec	1	1.00	1.00	2.76	1.00	2.76	3.84
72		2	2.00	1.96	3.63	1.96	3.94	17.19
73		4	3.96	3.77	3.96	3.79	3.79	32.44
74		8	7.84	7.01	4.89	7.08	4.49	50.62
75		16	15.44	12.44	6.58	12.65	5.43	67.15
76		32	29.56	20.75	8.66	21.22	6.76	76.63
77	apache.codec	1	1.00	1.00	11.53	1.00	11.53	16.77
78		2	1.99	1.95	10.54	1.95	12.06	41.73
79		4	3.91	3.70	10.26	3.72	13.86	62.34
80		8	7.62	6.74	9.96	6.85	13.76	73.86
81		16	14.57	11.67	9.74	11.97	12.02	75.26
82		32	26.81	19.32	9.23	20.04	10.72	70.08
83	apache.lang3	1	1.00	1.00	2.84	1.00	2.84	4.71
84		2	1.99	1.99	3.88	1.99	5.13	20.16
85		4	3.98	3.96	3.47	3.96	4.73	34.31
86		8	7.94	7.83	3.31	7.84	4.66	47.36
87		16	15.74	15.32	3.47	15.38	4.63	57.58
88		32	31.12	29.57	4.63	29.83	5.00	67.45
89	eclipse.draw2d	1	1.00	1.00	6.68	1.00	6.68	7.65
90		2	1.83	1.69	10.76	1.69	10.87	28.01
91		4	3.43	2.86	13.96	2.88	13.79	47.89
92		8	5.97	4.22	16.28	4.29	16.02	54.15
93		16	9.75	5.69	18.01	5.86	17.62	55.98
94		32	14.81	7.21	19.52	7.4	18.47	54.68
95	htmlparser	1	1.00	1.00	1.84	1.00	1.84	3.35
96		2	1.94	1.84	11.33	1.84	11.62	14.16
97		4	3.63	3.08	12.26	3.14	9.20	33.09
98		8	6.69	4.90	10.99	5.12	9.41	51.54
99		16	11.19	7.20	10.73	7.54	10.50	64.62
100		32	18.59	10.71	12.69	10.95	11.50	68.51
	AVG	10.50	7.52	5.19	8.06	5.37	8.25	34.96
	Overall AVG	7.30	5.51	3.69	9.997	3.79	10.001	30.61

Table A.9: MFL W-scores – Steimann Benchmarks

A.4 PFL Case Study

In this section of the appendix we present a case study to illustrate how PFL-PPV deals with a typical multiple-fault program from the Steimann benchmarks. The case study is presented for two reasons. Firstly, it is presented in order to illustrate a general trend in the Steimann benchmarks that faults were covered in failing traces by at least one test case that itself covered few components, thus advantaging fault localisation methods which satisfy the fault localisation properties presented in Chapter 6. Secondly, it is presented in order to illustrate a typical instance of a generated probability mass function as computed using the PFL-equations. We present the coverage matrix for eventbus-1357981291647 as an example in Table A.10. This example is in the set of 4-fault versions of the Steimann benchmarks, and contains three covered faults. It has 42 program components (labelled on the first row) six failing traces (the top six rows of the matrix), and 84 passing traces (in the remaining rows). The faults are C_{22}, C_{32}, and C_{39}. As we can observe from the matrix, these components form a faulty hitting set. When a fault is covered in a test case, we have put the **1** in bold. No fault covers all failing traces. For each component we present its probability, as computed by the PFL-equations, in Table A.1. No component receives a probability of 1. The probability of the faults are as follows. $P(h_{22}) = 0.98$, $P(h_{32}) = 0.46$, and $P(h_{39}) = 0.02$. The first two faults have the highest fault likelihood of any component. The last fault has an extremely low fault probability. The faults are represented in black in Table A.1.

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}	C_{21}	C_{22}	C_{23}	C_{24}	C_{25}	C_{26}	C_{27}	C_{28}	C_{29}	C_{30}	C_{31}	C_{32}	C_{33}	C_{34}	C_{35}	C_{36}	C_{37}	C_{38}	C_{39}	C_{40}	C_{41}	C_{42}	E
1	0	1	1	0	0	1	1	1	1	1	1	1	0	1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	0	1	**1**	1	1	1	1
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	**1**	0	0	1	0	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	**1**	0	1	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	1	0	0	1	1	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	1	0	0	1	0	1	1	0	0	1	0	1	0	0	**1**	1	1	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	**1**	0	1	1	0
1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	1	1	0	1	0	1	0	1	1	**1**	0	1	1	0
1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	**1**	1	1	1	0
1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	**1**	1	1	1	0
1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	0	1	**1**	1	1	1	0
0	0	1	1	0	0	1	0	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	1	1	0	1	**1**	1	1	1	0
1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	1	1	0	1	0	1	0	1	1	**1**	0	1	1	0
0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	0	0	0
1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	1	1	0	1	**1**	0	1	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**	0	1	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	0	0	**1**	1	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
1	1	0	1	1	1	0	0	1	0	0	1	0	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	1	0	1	1	0	1	0	0	1	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	1	0	1	1	1	0	1	1	0	0	1	0	0	1	1	1	0	1	1	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	**1**	1	0	0	0
0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	**1**	1	1	1	0
0	1	1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	**1**	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	1	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	**1**	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	**1**	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	**1**	0	1	1	0
0	1	0	0	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0
0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	**1**	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	**1**	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	1	0
0	1	1	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	1	0	**1**	1	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**1**	0	1	0	0
0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0

Table A.10: eventbus-1357981291647

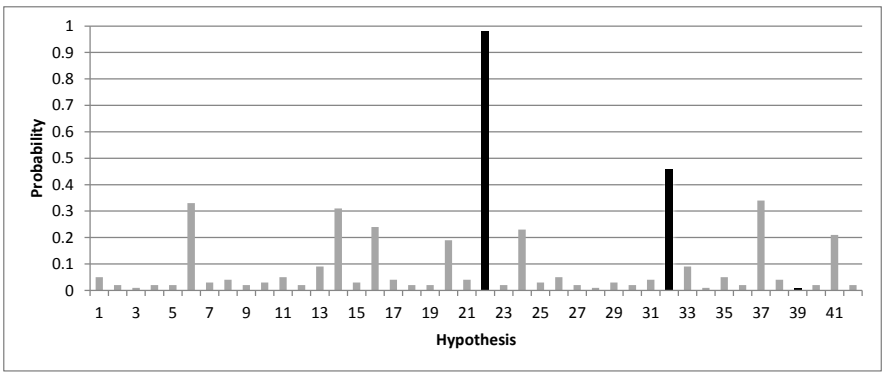

Figure A.1: Fault probabilities for eventbus-1357981291647