Software Debugging Using Program Spectra

by

Lee Hua Jie

Submitted in total fulfilment of
the requirements of the degree of

Doctor of Philosophy

September 2011

Department of Computer Science and Software Engineering
The University of Melbourne, Australia

To my parents, who support me in everything that I do.

Abstract

This thesis focuses on debugging using program spectra. Program spectra captures the dynamic behaviour of a program indicating which program statements are executed by respective test cases, which include pass and fail cases. By using this information, we use functions to rank all statements to locate bugs. Statements ranked top of the ranking are more likely to be buggy. We refer to these functions as spectra metrics. In a traditional debugging task, the programmer often has to examine program execution step-by-step within a block or function of program code that is most likely to be buggy. Using program spectra information can help the programmer to narrow down to those program statements that are most likely to be buggy.

The thesis contributes to the theoretical understanding of debugging single bug programs using program spectra. We propose several spectra metrics and also review other metrics suggested for bug localization. Some of the metrics that have been previously employed in other domain areas such as biological science have been adapted for the debugging area and their effectiveness have been evaluated. We propose several methods to help improve the precision of bug localization. We show by employing more information such as frequency execution and coverage of test cases can improve bug localization performance significantly.

Our extensive evaluation on several benchmarks, namely Siemens Test Suite, subset of Unix Test Suite, Concordance, and Space indicate that our proposed spectra metrics are effective in improving bug localization performance. This thesis work advances the state-of-the-art of bug localization and consequently has great potential to improve the effectiveness of debugging software.

Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of table, maps, bibliographies, appendices and footnotes.

Hua Jie Lee

Preface

This thesis is mostly based on original work jointly conducted with my advisors, Dr. Lee Naish and Prof. Kotagiri Ramamohanarao (Rao). The results of the research are presented in the following five peer reviewed papers:

- **Naish, L., Lee, H., and Ramamohanarao, K.** *A Model for Spectra-based Software Diagnosis*. In Volume 20 – Issue 3 of Transactions on Software Engineering and Methodology (TOSEM). ACM [Naish et al., 2011].
  - Most of the contents of this paper can be found in Chapter 5.
  - Extensive evaluation on other benchmarks such as the subset of the Unix Test Suite, Concordance, and Space using the proposed optimal metrics and other spectra metrics are also detailed in the chapter.
- **Lee, H., Naish, L., and Ramamohanarao, K.** *Study of the Relationship of Bug Consistency with respect to Performance of Spectra Metrics*. In Proceedings of the 2009 2nd International Conference on Computer Science and Information Technology, pages 501–508. IEEE Computer Society [Lee et al., 2009a].
  - The contents of this paper can be found in Chapter 6.
- **Lee, H., Naish, L., and Ramamohanarao, K.** *The Effectiveness of Using Non redundant Test Cases with Program Spectra for Bug Localization*. In Proceedings of the 2009 2nd International Conference on Computer Science and Information Technology, pages 127–134. IEEE Computer Society [Lee et al., 2009b].
  - The contents of this paper can be found in Chapter 7.
  - We further investigate the effectiveness of bug localization using larger number of unique pass and fail test cases in the chapter.
- **Naish, L., Lee, H., and Ramamohanarao, K.** *Spectral Debugging with Weights and Incremental Ranking*. In Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference, pages 168–175. IEEE Computer Society [Naish et al., 2009].
  - Most of the contents of this paper can be found in Chapter 8.
  - The chapter includes analysis of the proposed incremental ranking approaches on O and O^p metrics using our model program [Naish et al., 2011].

- **Lee, H., Naish, L., and Ramamohanarao, K.** *Effective Software Bug Localization Using Spectral Frequency Weighting Function*. In Proceedings of the 2010 34th Annual IEEE Computer Software and Applications Conference, pages 218–227. IEEE Computer Society [Lee et al., 2010].
  - Most of the contents of this paper can be found in Chapter 9.
  - Further evaluation to validate the effectiveness of the proposed approach on other single bug and multiple-bug programs, namely Concordance and Space, can also be found in the chapter.

I have also co-authored the following paper with my advisors:

- **Naish, L., Lee, H., and Ramamohanarao, K.** *Statements versus Predicates in Spectral Bug Localization*. In Proceedings of the 2010 17th Asia-Pacific Software Engineering Conference, pages 375–384. IEEE Computer Society [Naish et al., 2010].
  - This paper describes the study of using predicate-based spectra metrics to locate bugs effectively. Due to the lack of space in this thesis, the contribution of this paper is briefly described in Chapter 2.

Acknowledgements

First and foremost, I would like to thank my advisors, Dr Lee Naish and Professor Kotagiri Ramamohanarao. They always provide me advice and guidance throughout these years. Coming from the working background of software testing in the industry prior to this, I gained different perspectives of research in this area. When I am facing research problems, they never fail to give constructive suggestions which always lead me to think out of the box. Along this research journey, I am grateful to them that they always give me the flexibility to explore and pursue different angles to solve a particular research problem. From the constant meetings and discussions with my advisors, they trained me to possess the attributes of a researcher. Hence enabling me to discuss, present research ideas and findings confidently.

I would also like to thank my Ph.D. advisory committee member, Dr Tim Miller for his valuable feedback who has helped me tremendously to improve the quality of my thesis. I would like to thank all the other academic staff of the Department of Computer Science and Software Engineering, University of Melbourne. I have worked with all of them at some point during the past three-and-a-half years. I would also like to acknowledge the generous support of the University of Melbourne and the Department of Innovation, Industry, Science and Research in providing Endeavour International Postgraduate Research Scholarships (IPRS) for my research. I would like to extend my gratitude to Bernard Pope, William Webber, and Alex Stivala for proofreading the earlier draft of my thesis.

Thanks to my fellow students and colleagues at the University of Melbourne, in particular my previous and current office-mates, Adel, Andreas Schutt, Hairo, Lei Ni, Martin, Raj Gaire, Tanzima Hashem, and ZiYuan. To my friends Eunus, Fan, Jubaer Arif, Juni Ong, Kapil, Michelle, Mukaddim, Parvin, Rafiul, SriDevi, Stephen Ng, and Sue Lynn. I would also like to thank David Staples who never fail to help me whenever I face problems in computer and server configuration. He first introduced me to the world of Unix and Bash in my early days of my Ph.D. Thank you all for making the research experience at the University of Melbourne even more rewarding.

Last but not least, I would like to express my thanks to my family. My dad, mum, brother and sister have always been there for me and supported me. They always encourage me to pursue what I like in my life and never to give up easily.

Lee Hua Jie
Melbourne, Australia
September 2011.

Contents

List of Tables

List of Figures

List of Algorithms

GLOSSARY OF TERMS

Binary information of execution count is defined as the binary form of program spectra. The information is indicated by 1 and 0, for the statement being executed and not executed by a particular test.

Black box testing is defined as testing activity on program code which is subjected to inputs and its outputs are verified for conformance to specified behaviour [Beizer, 1995]. The software user is only concerned with functionalities and features. This is also known as functional testing [IEEE, 2004c].

Block is defined as a set of program statements where if any program statement in the set is executed, then all program statements contained in the respective set are executed as well.

Bug refers to a program statement that, when executed by test cases, has unintended behaviour. Fault refers to incorrect program statements defined in the program that cause program failure. Failure refers to the deviation of the observed behaviour of a program from its specification. For example, the output of a program returns the value of 2 instead of expected output value of 1 [IEEE, 2004a].

Bug localization performance refers to the effectiveness of respective bug localization approaches. Bug localization performance can be measured using different performance measures (refer to the details in Chapter 4).

Clustering is commonly known as division of data into groups of similar objects [Berkhin, 2006]. Previous studies attempted to use similarity measures [Tan et al., 2004], namely Euclidean distance, as a measurement of how similar the data is in a group.

Dynamic analysis defines the process of evaluating a system or component based on its behaviour during execution.

KLOC refers to a thousand lines of code.

Metric refers to spectra metrics. It refers to a numeric function and serves as a standard of measurement in terms of the likelihood of a program statement to be buggy.

Oracle refers to program code that is assumed not to contain bug. It is used to determine the correctness of respective test cases. It is also known as base version program code.

Performance Measure refers to a measure used to gauge the effectiveness of respective bug localization approaches.

Predicate, $Pred$, refers to a logical function evaluated at a decision (if-then-else), return values, and scalar pairs.

Program dependence graph (PDG) represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands), and the edges incident to a node represent both the data values on which the nodes operations depend and the control conditions on which the execution of the operations depends [Ferrante et al., 1987].

Rank percentages refers to the percentages of the program code to be examined in order to locate a bug in the program.

Single bug program refers to a program with one bug in it.

Spectra metrics refers to the formula used to predict which program statement is most likely to be buggy.

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [IEEE, 2004c].

System Dependence Graphs (SDG) is a collection of program dependence graphs (PDGs) [Ferrante et al., 1987, Beck and Eichmann, 2002].

Test case is defined as documentation that specifies inputs, predicted results, and a set of execution conditions for a test item [IEEE, 2004a].

White-box testing is testing that takes into account the internal mechanism of a system or component [IEEE, 2004c]. White-box testing is also known as structural testing, clear box testing, and glass box testing [Beizer, 1995]. It refers to the programmer having access to the internal workings of the software, especially the logic and structure of the program code.

NOTATION

Symbol	Definition
B	number of bugs or faults in program
P	correct version of a program
P′	a fault-seeded program
T	total number of test cases
t	test case
$totP$	number of pass test cases
$totF$	number of fail test cases
$e_{s,t}$	test execution coverage of statement s for test t
a_{ef}	number of fail test case(s) executing the particular statement
a_{ep}	number of pass test case(s) executing the particular statement
a_{nf}	number of fail test case(s) not executing the particular statement
a_{np}	number of pass test case(s) not executing the particular statement
a_{ij}	spectra properties of a_{ef}, a_{ep}, a_{nf}, and a_{np}, with $i \in \{n, e\}$ and $j \in \{p, f\}$
$MetValue$	metric value of particular statements evaluated with a given spectra metrics
S_p	set of statement(s)/block(s) executed by a pass test case
S_f	set of statement(s)/block(s) executed by a fail test case

1 Introduction

Software is widely used in various areas including the public sector, financial services, manufacturing, communications, retail, services, defence, and healthcare. Newman reported that the total sales of software reached approximately USD$180 billion in the year 2000 [Newman, 2002]. In a recent study conducted by Gartner, it was estimated that the end-users and worldwide enterprises spent USD$3372 billion and USD$225 billion for software in the fiscal year 2008 respectively [Gartner, 2010]. In 2001, Microsoft sold more than USD$5 billion worth of software in Europe, the Middle East, and Africa [Festa, 2001].

As the software evolves and the complexity of software grows, bugs [1] are introduced. In this thesis, a bug refers to a program statement that has unintended behaviour when executed by test cases. Several major incidents have occurred due to software bugs. In Sydney, the 7-year-old M5 tunnel has been closed six times and most recently twice in a month [ABC, 2008, Emerson, 2008]. The closure was due to a bug in the software that controls the fire and air-circulation systems, which led to a 3-hour shutdown, causing delays for motorists during peak hours. In Melbourne, Australia, Myki [myki, 2010] has been introduced as a smartcard ticketing system for public transport, similar to the Oyster [Oyster, 2010] in London. It has been reported that there was a software bug where two Myki users were credited with more than A$150K of credit to their accounts [ABC, 2010]. The company admitted that this was due to a bug in the fare calculation algorithm of the Myki system.

In October 2007, Tokyo's railway stations encountered more than 4000 faulty automatic gate machines [Williams, 2007]. These gates are meant for passengers accessing the railway using a contactless smart card. The failure was due to a bug in the automatic

[1] Bug or error has been used as early as the 1880s [Shapiro, 1987]. The etymology of the bug has been later claimed during the incident of a bug found in one of the circuits of the Harvard Mark II computer by the late US Navy Captain Grace Murray Hopper. It occurred when she was trying to fix a blocked relay in the computer in Harvard University in 1947. She found a two-inch moth which caused the relay to block and ever since then, the "bug" term has been used [Dennette A. Harrod, 1996].

gate machine software that caused an overflow of stolen card data being sent to the gate machines. This failure caused the gate machine system to crash and affected 2.6 million passengers. It took the software vendor more than half a day to locate and fix the bug, due to the high complexity of the program code. More details of other major software bug incidents that have happened around the world are listed in Appendix A.

The severity of the software bug incidents mentioned above indicates that it is very important to produce robust and reliable software. A typical software life cycle requires a feasible plan, detailed analysis, good design of requirements, implementation, testing, and maintenance [Everett and McLeod, 2007]. Testing is one of the most important tasks, and the analyst firm Pierre Audoin Consultants (PAC) reports that the software testing spending of worldwide organisations was €79 billion in the year 2010 [Nichols, 2010].

Software testing can be classified into two types; black box testing and white box testing. A typical software testing task involves identifying any bug that exists in the program [IEEE, 2004b]. The process of identifying and locating a bug is better known as bug localization, and this term is used throughout the thesis. Traditionally, the programmer debugs the program code step-by-step to locate the bug. Upon identifying the bug, the programmer fixes the program in order to ensure the software works according to the defined requirements.

The traditional debugging approach is usually performed by examining the program code in a debugger specific to the programming language. Experienced developers are usually able to guess the buggy region of the program code and take appropriate actions to fix the bug. However, this approach is not in general practical as software complexity increases. Identifying the buggy region of code is difficult even for experienced programmers, and can cost a great deal of time and wasted effort to locate the bug in the program code.

It has been estimated that debugging accounts for over 50 percent of the time spent in a typical programming project [Hailpern and Santhanam, 2002]. According to a report from the National Institute of Standards and Technology (NIST) [Newman, 2002], software faults (i.e., bugs or errors) cost the U.S. economy an estimated USD$59.5 billion annually, or approximately 0.6% of GDP in 2002.

Automated software debugging techniques have been proposed to substantially reduce the time spent in debugging (locating the bug). There are primarily two automated debugging approaches: static analysis, and dynamic analysis. Static analysis, originating in 1970, investigates properties of programs by analysing their source code alone, without recourse to their dynamic execution. The effectiveness of static analysis depends on the programming languages' computational model and type of data structure the language supports. However, static analysis has limited capability to capture bug behaviour; for

example, it cannot detect logical bugs. A concrete example is the application logic error in a banking system. A bank system should ensure that there is an overdraft limit based on each individual, which is applied when the customer withdraws money. If there is a bug in checking the limit of the overdraft of an individual in the system, static analysis is unable to detect such a bug. Dynamic analysis is the other debugging approach, which focuses on the execution of software code with different test cases (test inputs) to reveal potential software bugs. These test cases are often called test suites. Test suites have the common purpose of verifying that the software exhibits intended behaviour when executed.

A recent form of dynamic analysis is to collect program spectra [Reps et al., 1997]. In this approach, the program is executed by the test suite to gather information on the dynamic behaviour of the program. The information gathered indicates which program statements are executed by test cases (both pass and fail test cases). Using this information, we can apply the spectra metrics to locate bugs in the program. Zoeteweij et al. applied the program spectra approach in the embedded software of a consumer electronics product, namely teletext [Zoeteweij et al., 2007]. They showed that bugs can be located effectively using this approach. Recent studies in this area have focused on applying different spectra metrics for debugging [Abreu et al., 2006, Xie et al., 2010, Lucia et al., 2010,Naish et al., 2011]. These methods have shown to be effective in bug localization. In this thesis, we extensively study various spectra metrics, and propose several new spectra metrics to improve bug localization performance.

1.1 Contributions

We make the following contributions:

1. Propose a simple model program based on the if-then-else statements containing a single buggy statement to understand bug localization characteristics. Through this understanding, we develop optimal spectra metrics for single bug programs. We observe that the optimal spectra metrics of the simple model program in fact work even for real programs.

2. Show that two spectra metrics are equivalent in ranking if and only if any one of them can be made identical to the other using a monotonically increasing function (Lemma 5.2.1). Using this property, we establish several spectra metrics are equivalent.

3. Observe a significant number of test cases had identical test coverage in the existing test suites. Investigate the effectiveness of bug localization performance using the

test cases in the entire test suites and unique test cases (that is considering only one of the test cases from all the test cases that have identical test coverage). We show no degradation of bug localization performance using unique test cases for better performing metrics on most of our benchmarks. We also show that by using a larger number of unique (non-redundant) pass and fail test cases improves bug localization performance significantly.

4. Assign different weights to the respective fail test cases according to how informative the fail tests are. We compare the effectiveness of bug localization performance using the proposed approach with that of not assigning any weights. We show significant improvement in bug localization performance with our proposed top-down incremental approach, especially on multiple-bug programs for better performing metrics, with an improvement of average rank percentages ranging from 0.41% to 4.63%.

5. Existing bug localization approaches use the binary form of test coverage (whether each statement is executed or not by each test case) to locate bugs. We propose to use more information from test coverage by using frequency counts [Liu et al., 2005], that is, the number of times each statement is executed. We propose using a sigmoid function to map frequency counts to the respective spectra properties. Using this approach, we show improvement in bug localization performance for most of the better performing metrics as compared to using the binary approach especially on multiple-bug programs. The latter improvement is in the range of average rank percentages from 0.03% to 4.52%.

1.2 Structure of the thesis

Chapter 2 details the background and several software bug localization studies that are closely related to the thesis. Chapter 3 presents a detailed survey of software fault localization techniques. Chapter 4 discusses some of the existing performance measures used in debugging and propose several new performance measures. Chapter 5 describes the approach of using a simple model program (if-then-else) to understand the behaviour of single bug programs. We develop optimal spectra metrics which locate bugs in single bug programs effectively. In Chapter 6, we describe the relationship between bug consistency and bug localization performance. Chapter 7 details bug localization performance of using unique (non-redundant) and redundant test cases. This chapter also describes the importance of having a larger number of unique (non-redundant) test cases that can improve bug localization performance. We describe different weights assigned according

to how informative respective fail tests are in Chapter 8. Chapter 9 describes the approach of using frequency counts, which incorporates more information of test coverage in order to locate bugs effectively. In this chapter, we describe how a sigmoid function can be used to map the frequency coverage to respective spectra properties. Chapter 10 concludes the thesis with some discussions of our proposed approaches and possible future directions.

2

Background and Literature Review

2.1 Introduction

In software bug localization, there are mainly two types of analysis: static analysis and dynamic analysis. In the thesis, we are particularly interested in dynamic analysis.

Dynamic analysis is always performed through program execution. In practice, numerous test cases will have been written to ensure program requirements have been met. Test case failure leads to the discovery of bugs in the program code. Code coverage information can be captured using dynamic analysis. The analysis indicates the extent of the program code that has been executed by the test cases. Different types of code coverage can be used in dynamic analysis to locate potential bugs in the program code. These types are statements, blocks, functions, predicates, and paths of programs.

One of the advantages of using dynamic analysis, rather than static analysis for locating bugs is the ability to detect dependencies in the program code. Dynamic analysis also deals with the runtime values of the program code. However, often it is difficult to assure full test coverage for program code using dynamic analysis. Performance is also compromised using dynamic analysis due to the instrumentation of the program code.

Several bug localization approaches using dynamic analysis have been proposed in the literature. Program spectra is one such approach. Initially, we introduce the concept of program spectra. We give a comprehensive list of the spectra metrics used in the thesis. We also detail the literature review of spectral debugging. This includes spectra-based approaches with respect to the different coverage types: statement-based spectra coverage, block-based spectra coverage, function-based spectra coverage, branch-based spectra coverage, and predicate-based spectra coverage. These approaches have been studied extensively to locate potential bugs in the program code.

Table 2.1: Example of Test Coverage Information (frequency counts) with Tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5
S_1	60	2	100	0	38
S_2	70	65	90	0	45
S_3	25	35	0	4	0
S_4	80	30	0	42	0
S_5	42	0	37	48	81
S_6	0	59	0	0	17
Test Result	Fail	Fail	Fail	Pass	Pass

Table 2.2: Example of Test Coverage Information (binary) and Program Spectra with Tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	0	1	1	0	1	3
S_2	1	1	1	0	1	1	0	1	3
S_3	1	1	0	1	0	1	1	1	2
S_4	1	1	0	1	0	1	1	1	2
S_5	1	0	1	1	1	0	1	2	2
S_6	0	1	0	0	1	1	2	1	1
				⋮					
Test Result	Fail	Fail	Fail	Pass	Pass				

2.2 Background

Program spectra [Reps et al., 1997] is a typical approach in dynamic analysis. It consists of information on the parts of a program that were executed during the test case executions. These parts could be individual statements, basic blocks, branches, or larger regions such as functions. In our study, we use individual statements. Using statements is equivalent to considering basic blocks, assuming normal termination (a statement within a basic block is executed if and only if the whole basic block is executed). During the execution of each test case, data is collected indicating statements that are executed. Additionally, each test case is classified as a *pass* or a *fail*. As the program spectra is used for debugging purposes, this method is also referred to as spectral debugging.

In program spectra, four numbers are ultimately produced for each statement, namely the number of pass/fail test cases in which the statement was/wasn't executed. We adopted the notation of [Abreu et al., 2006] to compute the a_{ij} values, which refer to the notation $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$. The first subscript indicates whether the statement was executed (e)

or not (n), and the second subscript indicates whether the test passed (p) or failed (f). For example, a_{ep} of a statement refers to the number of pass test cases which have executed the statement. Note that statement s is implicit in a_{np}, a_{nf}, a_{ep}, and a_{ef} throughout the thesis.

Table 2.1 is the matrix of an example of raw program test coverage information $e_{s,t}$ with one row for each program statement s and one column for each test case, t. The test coverage information of Table 2.1 consists of frequency counts. This table states the number of times each program statement has been executed by the respective test cases. Table 2.2 refers to the binary form of the program spectra of Table 2.1. The table states whether each program statement has been executed or not by the respective test cases (regardless of the number of times it has been executed). Each cell of this table indicates whether a particular statement is executed (the value is 1) or not (the value is 0) for a particular test case. We refer to the test coverage as spectra coverage. Additionally, there is a vector with values indicating the $Pass$ or $Fail$ of each test case. In this example, both tables contain six statements and a total of five tests. We use the term $totP$, $totF$, and T to represent the total number of pass, fail test cases, and the total number of tests of the program respectively. In this case, $totP$ is 2, and $totF$ is 3.

For each statement, we apply a *function* to map the four a_{ij} values to a single number. We call such *function* spectra metric (distance). In the thesis, *metric* is not related to the metric spaces where it defines the distance between any two points in the space. Metric is loosely used as a numeric function and serves as a standard of measurement of the likelihood of program statement being buggy. In Table 2.2, we observed the mapping of all the statements to the respective a_{ij} values. As Statement 1 (S_1) is executed by three fail tests; T_1, T_2, and T_3, therefore the a_{ef} of S_1 is 3. The a_{nf} of S_1 is simply 0 since all the three fail tests have executed S_1. The single number is known as the metric value, $MetValue$, which is evaluated with the respective spectra metrics for all the statements. These statements are ranked in descending order according to the $MetValue$, starting from the statement that has the largest $MetValue$. The higher the $MetValue$ a statement has, the more likely the statement is to be buggy.

2.3 Spectra Metrics

A comprehensive list of spectra metrics is given in Table 2.3. To date, 16 spectra metrics have been proposed in the debugging literature to predict the statements most likely to be buggy. These metrics are shown in the top of the table; details of the debugging literature can be found in Section 2.4. In the thesis, we contribute several new spectra metrics such

as O, O^p, JacCube, Ample2, Binary, and Wong3′. We defer the discussion of these metrics to Chapter 5.

Table 2.3: List of Spectra Metrics

Name	Formula	Name	Formula
O	-1 if $a_{nf} > 0$, otherwise a_{np}	O^p	$a_{ef} - \frac{a_{ep}}{totP+1}$
Jaccard	$\frac{a_{ef}}{totF+a_{ep}}$	Ochiai	$\frac{a_{ef}}{\sqrt{totF(a_{ef}+a_{ep})}}$
Tarantula	$\frac{\frac{a_{ef}}{totF}}{\frac{a_{ef}}{totF}+\frac{a_{ep}}{totP}}$		
Zoltar	$\frac{a_{ef}}{totF+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$		
Ample	$\left\lvert \frac{a_{ef}}{totF} - \frac{a_{ep}}{totP} \right\rvert$	Ample2	$\frac{a_{ef}}{totF} - \frac{a_{ep}}{totP}$
Wong1	a_{ef}	Wong2	$a_{ef} - a_{ep}$
Wong3	$a_{ef} - h$, where $h = \begin{cases} a_{ep} & \text{if } a_{ep} \leq 2 \\ 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ 2.8 + 0.001(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$		
Wong3′	$a_{ef} - h$, where $h = \begin{cases} -1000 & \text{if } a_{ep} + a_{ef} = 0 \\ Wong3 & \text{otherwise} \end{cases}$		
CBI Inc	$\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{totF}{T}$	CBI Log	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\log totF}{\log a_{ef}}}$
CBI Sqrt	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\sqrt{totF}}{\sqrt{a_{ef}}}}$	M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$
M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	M3	$\frac{2*(a_{ef}+a_{np})}{T}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$		
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$		
Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{totF} + \frac{a_{ef}}{a_{ef}+a_{ep}}\right)$		
Russell and Rao	$\frac{a_{ef}}{T}$	Lee	$a_{ef} + a_{np}$
Rogers & Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Simple Matching	$\frac{a_{ef}+a_{np}}{T}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{T}$
Hamming	$a_{ef} + a_{np}$	Euclid	$\sqrt{a_{ef} + a_{np}}$
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}}$		
Ochiai3	$\frac{a_{ef}^2}{totF(a_{ef}+a_{ep})}$		
Platetsky-Shapiro	$a_{ef} + a_{ef}^2 + totF - a_{ep}a_{ef} - a_{ep}a_{nf}$		
Collective Strength	$1 - \frac{a_{ef}+a_{np}}{(a_{ef}+a_{ep})(totF)+(a_{nf}+a_{np})(totP)} * \frac{1-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}{1-a_{ef}-a_{np}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(totF)(totP))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}$		

Continued on next page

Table 2.3 – continued from previous page

Name	Formula	Name	Formula
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(totF)(totP)}$		
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(totP)+(a_{nf}+a_{np})(totF)}$		
Scott	$\frac{4(a_{ef}a_{np}-a_{nf}a_{ep})-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4(a_{ef}a_{np}-a_{nf}a_{ep})-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$		
Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{totF}+\frac{a_{np}}{totP}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$		
Binary	0 if $a_{nf}>0$, otherwise 1	Gower1	$\frac{a_{ef}-(a_{nf}+a_{ep})+a_{np}}{T}$
Gower2	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+0.5*(a_{nf}+a_{ep})}$	Gower3	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{a_{ef}a_{np}+a_{nf}a_{ep}}$
Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$		
AddedValue	$\frac{a_{ef}}{\max(a_{ef}+a_{ep},totF)}$	Interest	$\frac{a_{ef}}{(a_{ef}+a_{ep})totF}$
Confidence	$\max\left(\frac{a_{ef}}{a_{ef}+a_{ep}},\frac{a_{ef}}{totF}\right)$		
Certainty	$\max\left(\frac{a_{ef}}{a_{ef}+a_{ep}}-(a_{ef}+a_{ep}),1-(a_{ef}+a_{ep})\right)$		
Sneath & Sokal 1	$\frac{2(a_{ef}+a_{np})}{T+a_{ef}+a_{np}}$		
Sneath & Sokal 2	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$		
Phi	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(totF)(a_{ef}+a_{ep})(a_{nf}+a_{np})(totP)}}$		
Kappa	$1-\frac{a_{ef}+a_{np}-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}{1-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}$		
Conviction	$\max\left(\frac{(a_{ef}+a_{ep})(totP)}{a_{ep}},\frac{(a_{nf}+a_{np})(totF)}{a_{nf}}\right)$		
Mountford	$\frac{2a_{ef}}{2(a_{ef}+a_{ep})(totF)-(2a_{ef}+a_{ep}+a_{nf})a_{ef}}$		
Klosgen	$\sqrt{a_{ef}}*\max\left(\frac{a_{ef}}{a_{ef}+a_{ep}}-totF,\frac{a_{ef}}{totF}-(a_{ep}+a_{ef})\right)$		
YuleQ	$\frac{a_{ef}a_{np}-a_{ep}a_{nf}}{a_{ef}a_{np}+a_{ep}a_{nf}}$	YuleY	$\frac{\sqrt{a_{ef}a_{np}}-\sqrt{a_{ep}a_{nf}}}{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}}$
YuleV	$\frac{a_{ef}a_{np}-(a_{ef}+a_{ep})(totF)}{a_{ef}a_{np}+(a_{ef}+a_{ep})(totF)}$		
Correlation	$\frac{\left\lvert a_{ef}a_{np}-(a_{ef}+a_{ep})(totF)\right\rvert-\frac{T}{2}}{\sqrt{(2a_{ef}+a_{ep})(totF+a_{np})(totF+a_{ef})(totP+a_{ef})}}$		
Manhattan	$1-\frac{a_{ep}+a_{nf}}{T}$	Braun	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
Baroni	$\frac{\sqrt{a_{ef}a_{np}}+a_{ef}}{\sqrt{a_{ef}a_{np}}+totF+a_{ep}}$	Coef	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
Levandowsky	$\frac{a_{ef}}{totF+a_{ep}}$	Watson	$1-\frac{(a_{nf}+a_{ep})}{2a_{ef}+a_{nf}+a_{ep}}$
JacCube	$\frac{a_{ef}}{\sqrt[3]{totF+a_{ep}}}$	NFD	$a_{ef}+a_{np}$
SokalDist	$\sqrt{\frac{a_{ef}+a_{np}}{T}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$
CorRatio	$\frac{a_{ef}^2}{totF(a_{ef}+a_{ep})}$	Forbes	$\frac{Ta_{ef}}{totF(a_{ef}+a_{ep})}$
Fager	$\frac{a_{ef}}{\sqrt{totF(a_{ef}+a_{ep})}}-\frac{1}{2\sqrt{a_{ef}+a_{ep}}}$		

Continued on next page

Table 2.3 – continued from previous page

Name	Formula	Name	Formula
McConnaughey	$\frac{a_{ef}^2 - a_{nf}a_{ep}}{(totF)(a_{ef}+a_{ep})}$	Simpson	$\frac{a_{ef}}{totF}$
AssocDice	$\frac{a_{ef}}{\min((a_{ef}+a_{ep}),totF)}$	Dice	$\frac{2a_{ef}}{totF+a_{ep}}$
Fossum	$\frac{T(a_{ef}-0.5)^2}{(totF)(a_{ef}+a_{ep})}$		
Pearson	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(totF)(a_{ef}+a_{ep})(a_{nf}+a_{np})(totP)}}$		
Dennis	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(T)(totF)(a_{ef}+a_{ep})}}$		

Definition 1 (Spectra Metric J-Measure (J-Meas)). $a_{ef} * log(\frac{a_{ef}}{totF*(a_{ef}+a_{ep})}) + \max(a_{ep} * log(\frac{a_{ep}}{totP*(a_{ef}+a_{ep})}), a_{nf} * log(\frac{a_{nf}}{totF*(a_{nf}+a_{np})}))$

Definition 2 (Spectra Metric Wong4). $[(1.0)*n_{F,1} + (0.1)*n_{F,2} + (0.01)*n_{F,3}] - [(1.0)*n_{S,1} + (0.1)*n_{S,2} + 0.0001 * \frac{totF}{totP} * n_{S,3}]$

$$
\begin{aligned}
where \quad n_{F,1} &= \begin{cases} 0, & \textit{for } a_{ef} = 0 \\ 1, & \textit{for } a_{ef} = 1 \\ 2, & \textit{for } a_{ef} \geq 2 \end{cases} \\
n_{F,2} &= \begin{cases} 0, & \textit{for } a_{ef} \leq 2 \\ a_{ef} - 2, & \textit{for } 3 \leq a_{ef} \leq 6 \\ 4, & \textit{for } a_{ef} > 6 \end{cases} \\
n_{F,3} &= \begin{cases} 0, & \textit{for } a_{ef} \leq 6 \\ a_{ef} - 6, & \textit{for } a_{ef} > 6 \end{cases} \\
n_{S,1} &= \begin{cases} 0, & \textit{for } n_{F,1} = 0, 1 \\ 1, & \textit{for } n_{F,1} = 2 \textit{ and } a_{ep} \geq 1 \end{cases} \\
n_{S,2} &= \begin{cases} 0, & \textit{for } a_{ep} \leq n_{S,1} \\ a_{ep} - n_{S,1}, & \textit{for } n_{S,1} < a_{ep} < n_{F,2} + n_{S,1} \\ n_{F,2}, & \textit{for } a_{ep} \geq n_{F,2} + n_{S,1} \end{cases} \\
n_{S,3} &= \begin{cases} 0, & \textit{for } a_{ep} < n_{S,1} + n_{S,2} \\ a_{ep} - n_{S,1} - n_{S,2}, & \textit{for } a_{ep} \geq n_{S,1} + n_{S,2} \end{cases}
\end{aligned}
$$

Over eighty spectra metrics have been proposed in domains, such as botany, zoology, biometrics, and data mining for the purpose of classification. As metrics are rapidly introduced in different domains, some of these metrics are identical. For example, the Rogot2 metric has been proposed to classify two different methods (e.g different questionnaires) to diagnose disease conditions. This metric is identical to the Sokal1 metric, which is used in the numerical taxonomy [Sokal and Sneath, 1963].

The oldest of the metrics is Jaccard, originally used for classification in the botany domain [Jaccard, 1901]. It has been used in other areas such as data mining and machine learning [Tan et al., 2002]. In the botany domain, several other metrics have also been been used, such as Russell and Rao (Russell), Sørensen-Dice, Dice, and Rogers & Tanimoto (Rogers) [Russel and Rao, 1940, Sørensen, 1948, Duarte et al., 1999, Rogers and Tanimoto, 1960]. In these studies, the metrics are used to classify species of plants. The Sørensen-Dice metric is widely known as Czekanowski [Bloom, 1981]. Braun was also originally used in the field of botany [Braun-Blanquet, 1932].

The other domain that uses spectra metrics is zoology. Mountford and Number of Features of Difference (NFD) were originally used in zoology [Mountford, 1962, Stephenson et al., 1968]. Ochiai is used [Ochiai, 1957] for analysing the ecological relationships (clusters) of different species of fishes in Japan. Forbes and Simpson were also originally used in zoology [Forbes, 1933, Simpson, 1961]. Fager and McConnaughey (McCon) were used in the study of plankton [Fager and McGowan, 1963, McConnaughey and Laut, 1964]. Coefficient differences (Coef) was originally used to study the distribution of amphibians and reptiles [Peters, 1968, Savage, 1960].

CorRatio was used in the field of marine biology [Sorgenfrei, 1958]. Simple Matching comes from the area of biology [Kaesler, 1966]. Baroni-Urbani has been used in the study of ants [Urbani, 1976]. Levandowsky et al. proposed to use the inverse of the Jaccard metric to measure the dissimilarity between two sets of data in the area of plant biology [Levandowsky and Winter, 1971, Levandowsky, 1972].

Anderberg was originally proposed as a similarity measure in clustering objects [Anderberg, 1973]. In the field of clustering using the Self-Organizing Maps (SOM) algorithm [Kohonen, 2002], other similarity measures (metrics) have been introduced and evaluated, namely Kulczynski1, Kulczynski2, Hamann, and Sokal [Lourenco et al., 2004]. Hamann was originally introduced in the study of botany [Hamann, 1961]. Jaccard and Simple Matching have also been used in clustering, along with three unnamed metrics that we refer to as M1, M2, and M3 respectively [Everitt, 1978]. Watson was originally used as a similarity measure in the area of clustering data [Lance and Williams, 1966].

The Phi metric is used in statistics [Udny Yule and Kendall, 1948], specifically for chi-square, χ^2 [Greenwood and Nikulin, 1996]. The Kappa metric was originally used to measure the agreement of pairs of variables [Cohen, 1960]. Hamming was originally introduced for error detecting/correcting codes [Hamming, 1950]; for binary numbers it is equivalent to Lee [Lee, 1958] and Manhattan [Krause, 1973]. The Lee metric is used to compute the distance between two strings of same length. This metric is applied in the area of phase modulation [Berlekamp, 1968]. Manhattan and Euclid have been used in clustering [Krause, 1973]. Gower introduced the Gower1, Gower2, and Gower3 met-

rics based on the Euclidean properties [Gower, 1971, Gower and Legendre, 1986, Everitt and Rabe-Hesketh, 1997]. Fossum was originally used in the study of similarity of the chemical components [Holliday et al., 2003]. In the area of biometrics, various metrics have been introduced: Goodman and Kruskal [Goodman and Kruskal, 1954], Scott [Scott, 1955], Fleiss [Fleiss, 1965], Cohen [Cohen, 1960], Geometric Mean (GMean) [Maxwell and Pilliner, 1968] as well as Arithmetic Mean (AMean), Harmonic Mean (HMean), Rogot1, and Rogot2 [Rogot and Goldberg, 1966].

AddedValue [Sahar and Mansour, 1999], Certainty, CollectiveStrength (CollectiveS), Platetsky-Shapiro, Confidence, Interest, and Conviction have been used in the data mining communities [Tan et al., 2002]. There are other metrics used as association measures, such as Klosgen, J-Measure (J-Meas), Kappa, YuleY, and YuleQ. Metrics such as YuleY and YuleQ have been proposed in the data mining community to associate different attributes of any two variables [Yule, 1900]. J-Measure (J-Meas) is an index to study the probability of the distribution of variables (see Definition 1) [Smyth and Goodman, 1991]. Klosgen was originally used in the knowledge discovery system, Explora [Klosgen, 1992].

Some of these metrics have been adapted in other domains such as information retrieval, molecular biology, and numerical taxonomy. Jaccard, Dice, Overlap, and Ochiai (a more general version of the Ochiai metric called Cosine) have been used in the field of information retrieval [Dunham, 2002]. Anderberg and Simple Matching metrics have been used in molecular biology [Meyer et al., 2004] to study the different relationships of maize species. Simple Matching and SokalDist (commonly known as Sokal Distance) were originally introduced in numerical taxonomy [Sokal and Michener, 1975].

Even though all the above-mentioned metrics have been used in different domain areas, they share an identical goal, which is to associate or classify data of similar characteristics. Such data is also known as clusters. In the debugging area, our goal is to find clusters of correct statements (with low metric values) and buggy statements (with high metric values). Here we give examples of the intuition behind some of these metrics. Jaccard devised a way to measure the similarity of two sets. It is the size of their intersections divided by the size of their union. We can apply this idea to spectral debugging by using the set of test cases that fails and the set of test cases for which a particular statement is executed. A second approach is to view a row of the matrix in Table 2.2 as a vector in n-dimensional space. The cosine of the angle between this vector and the vector of test results is another measure of similarity — this is what the Ochiai metric computes. A third way is to think of the rows and results of the table as bit strings. The number of bits that differ in two strings is a measure of dis-similarity (the Hamming distance). By first taking the complement of one of the bit strings, we obtain a measure of similarity — this is what our formula for the Hamming metric computes.

Most of these proposed metrics are closely related to *norms* of the associated *metric space* (our use of the term *metric* is not related to metric spaces in any precise technical sense). A norm is a measure of the distance between two points: it must be positive for distinct points, symmetric ($dist(a, b) = dist(b, a)$) and must satisfy the triangle inequality ($dist(a, c) \leq dist(a, b) + dist(b, c)$). Given a norm, statements (rows in the matrix) can be ranked according to how similar or close they are to the result vector (whether each test passes or fails). Some of the spectra metrics we use are referred to as norms in the literature; others are called measures or distances. Some are defined in more general cases, such as where we have a matrix and vector of arbitrary numbers. We restrict our attention to the special case of binary numbers where the data is summarised by the four a_{ij} values in Table 2.2. This introduces the equivalence of some spectra metrics that are distinct in more general cases (see Subsection 5.2.1).

Some of these metrics use the properties of statements not executed by fail tests, a_{nf} and statements executed by pass tests, a_{ep}. In debugging, the property of statements executed by fail tests is an important indication of the bug. More fail tests executing the statement indicates the likelihood of the statement to be the bug. A statement that is executed by more pass tests indicates that it is less likely the statement is the bug. Therefore, we adapt some of these metrics to use a_{ef} and a_{np} instead of a_{nf} and a_{ep} respectively. For example, the NFD metric is originally $a_{nf} + a_{ep}$ [Stephenson et al., 1968]. Having a_{nf} and a_{ep} as the numerator of the metric cannot indicate statements that are likely to be buggy. Therefore we adapt this metric by inversing the existing metric. Another example is the Levandowsky metric [Levandowsky and Winter, 1971], which refers to the inverse of the Jaccard metric. The inverse of the Jaccard metric uses a_{nf} and a_{ep}. Therefore, we adapt Levandowsky to the Jaccard metric. Similar adaptation is made for other metrics, namely CollectiveS, Kappa, Manhattan, Watson, and SokalDist.

2.4 Literature Review of Spectral Debugging Using Dynamic Analysis

There are several existing studies of spectral debugging which are closely related to the thesis. They are Tarantula [Jones et al., 2001], Pinpoint [Chen et al., 2002], Nearest Neighbour [Renieres and Reiss, 2003], Ample [Dallmeier et al., 2005], CBI [Liblit, 2004], SOBER [Liu et al., 2005] and other studies by Abreu et al. [2006], Wong et al. [2007], and Wong et al. [2009]. Jones et al., Renieres et al., and Wong et al. use statement coverage [Jones et al., 2001, Renieres and Reiss, 2003, Wong et al., 2007, Wong et al., 2010], Abreu et al. use block coverage [Abreu et al., 2006], Ample uses func-

tion coverage [Dallmeier et al., 2005] while CBI [Liblit, 2004] and SOBER [Liu et al., 2005] use predicates respectively. RAPID and HOLMES use branch and path coverage respectively [Hsu et al., 2008, Chilimbi et al., 2009]. These studies use different spectra information such as statement-based spectra coverage, block-based spectra coverage, function-based spectra coverage, predicate-based spectra coverage, branch-based spectra coverage, and path-based spectra coverage. These are all essentially program spectra with respect to different types of code coverage. For example, statement-based spectra coverage refers to the program spectra of statement coverage.

In this thesis, we report the bug localization performance of the above proposed approaches. The bug localization performance of an approach refers to the effectiveness of the proposed approach in locating bugs. Different performance measures have been used to compare the effectiveness of locating bugs using respective proposed bug localization approaches. We defer the discussion to Chapter 4.

In dynamic analysis, the program code is instrumented with respect to the coverage types (e.g statements, blocks, predicates, and functions, to name a few) before executing test cases over the instrumented program code. Most studies instrumented the program code using `gcov` [Renieres and Reiss, 2003, Jones and Harrold, 2005, Abreu et al., 2006, Xie et al., 2010]. Wong et al. use the χSuds [Telcordia Technologies, Inc., 1998] to instrument the program code [Wong et al., 2007, Wong et al., 2010]. These studies rely on an *oracle* to determine the correctness (Pass or Fail) of the test cases. CBI [Liblit et al., 2005] relies on program crashes to indicate Fail for the test case. Otherwise, the test case is Pass. Most studies use a base version program code as the *oracle*. Base version program code refers to program code that does not have any seeded fault and is assumed to be correct. For each test input to the base version program code, the expected output of the program is produced. Fault-seeded program code refers to program code that has been deliberately seeded with a bug.

The output from the test case executions on the fault-seeded version of the program code is compared with the output of the test case executions on the base version program code. If there are differences in the output between the base version and the fault-seeded version for a particular test case, the test case is labelled as Fail. Otherwise, the test case is labelled as Pass.

Our benchmarks, such as the Siemens Test Suite, Space [Do et al., 2005], and a subset of Unix Test Suite [Wong et al., 1998], have the base version program code. Benchmarks such as Siemens Test Suite and Space programs are available in the Software-artifact Infrastructure Repository (SIR) [SIR, 2010]. These programs are provided for the purpose of evaluating the effectiveness of different approaches to locate bugs. In this repository, most of the C program benchmarks have base version program code. In practice, the base

version program code that serves as an *oracle* does not necessarily exist. There exists only the program code with the bug. For each test input to the program code, there are several approaches used to determine the expected output, such as program specifications and documentation.

The output of dynamic analysis is the test coverage information labelled by test cases (see Table 2.1 and Table 2.2). The matrices in these tables are known as the *spectra coverage*, and this term is used throughout the thesis.

Program spectra was introduced by Reps et al. [1997] to resolve the problem of the Y2K (year 2000) bug [Thomsett and Co, 1998]. Reps et al. [1997] proposed using the path-based spectra coverage approach where the path spectrum for each test case execution is compared. They compared the test case executions on the program code before and after the year 2000 respectively. The differences can be used to identify the path(s) that caused the Y2K problem. In their study, they suggest several approaches to instrument the paths of the program; these approaches are listed below:

1. At the source-code (program code) level

2. As part of compilation by using intermediate representations

3. As object-code-level transformation by modifying the object-code files

4. As post-loader transformation by modifying the executable files

Jones et al. pioneered the development of a software visualisation debugging tool, Tarantula, to distinguish bugs in the program, particularly for imperative language [Jones et al., 2001]. They instrumented the program code at the statement level. In order to gather statement-based spectra coverage for the program code, they used test suites to execute the program code.

Table 2.4, taken from [Jones et al., 2002], shows the example of a simple program segment executed by several pass and fail test cases. The Pass and Fail status of the test cases are represented with P and F respectively. Jones et al. [2002] propose two approaches to distinguish the statements likely to be buggy. The first, the *discrete approach*, uses a three-colour system (Red, Green, and Yellow) to visualise statements. Statements that are only executed by pass test case(s) are visualised as Green. Statements are visualised as Red if they are only executed by fail test case(s). Statements executed by both pass and fail test cases are visualised as Yellow. They found that this approach is not useful, as bugs could not be distinguished if the particular statement is executed by both pass and fail test cases.

Table 2.4: Mid Program from Jones et al. [2002]

		Test Cases					
		t1 3,3,5	t2 1,2,3	t3 3,2,1	t4 5,5,5	t5 5,3,4	t6 2,1,3
	`mid() { int x,y,z,m;`						
1:	`read("Enter 3 numbers:",x,y,z);`	•	•	•	•	•	•
2:	`m = z;`	•	•	•	•	•	•
3:	`if (y < z)`	•	•	•	•	•	•
4:	`if (x < y)`		•				
5:	`m=y;`		•				
6:	`else if (x < z)`	•				•	•
7:	`m=y; // *** bug ***`	•					•
8:	`else`	•		•	•		
9:	`if (x > y)`			•			
10:	`m=y;`			•			
11:	`else if (x > z)`						
12:	`m=x;`						
13:	`print("Middle number is:",m);`	•	•	•	•	•	•
	`}` Pass/Fail Status	P	P	P	P	P	F

Definition 3 (Colour Visualisation using the Continuous Approach for a statement s)**.**

$$colour = \textit{low colour (red)} + (\frac{\frac{a_{ep}}{totP} * 100\%}{\frac{a_{ep}}{totP} * 100\% + \frac{a_{ef}}{totF} * 100\%}) * \textit{colour range}$$

Definition 4 (Brightness for a statement s)**.**

$$bright = max(\frac{a_{ep}}{totP} * 100\%, \frac{a_{ef}}{totF} * 100\%)$$

The second approach, the *continuous approach*, is used to map the colour of respective program statements using the following Definition 3. It defines the *low colour*, which is red, as one end of the colour spectrum, and the *colour range* as the other higher end of the colour spectrum. Jones et al. [2002] use a graphical program such as Gimp [gimp, 2010] to define the values of the colours. As the *colour range* ranges from 0 (red) to 100 (green), it is defined as 100. The colour for a particular statement s would be a single number, representing a colour in the colour spectrum. In this definition, they calculated the proportion of the number of pass test cases that executed each statement as a fraction of the total number of pass and fail test cases executing that statement. These proportions are represented in terms of percentages. This proportion forms the complement of one

of the spectra metrics, Tarantula (see the metric in Table 2.3). We could possibly apply any other spectra metrics (see the metrics in Table 2.3) in the visualisation debugging tool. They also introduce a brightness component in the scale of 0 to 100 (Definition 4). A statement that is executed frequently, whether by pass or fail test cases, is assigned a brighter colour. Statement that is executed less frequently has a darker colour assigned.

If most pass test cases execute a program statement, the mapping colour shifts towards GREEN. If most fail test cases execute a statement, the mapping colour shifts towards RED. If a program statement has been executed by a similar amount of pass and fail test cases, the colour is mapped towards YELLOW. The intuition of the proposed mapping colour is to provide clear colour mixtures representing how often the program statement is executed by pass and fail test cases respectively. This helps to indicate the likelihood that each statement of the program is to be buggy.

Jones et al. [2002] use Space [Do et al., 2005] as their benchmark. This benchmark contains 20 single bug versions of 1000 test suites. The test suites consist of randomly selected test cases from the existing Space test suite. They also create multiple-bug versions of Space. In order to do that, they randomly select more than one single bug versions of Space and combine them as a multiple-bug program. The selected single bug program versions must contain different bugs in order to form the multiple-bug versions of Space. We use a similar approach to generate multiple-bug versions of our benchmarks (refer to Section 4.5). Jones et al. [2002] observe the Tarantula tool cannot distinguish multiple bugs in the program by using the colour spectrum. Therefore, this approach is only useful to locate single bug programs.

Jones et al. [2005] extend the previous study of Jones et al. [2002] by proposing the complement of Definition 3 (referred to as Suspiciousness in their study) as a metric to locate bugs. We refer to this as the Tarantula metric (refer this metric to Table 2.3). They use a similar statement-based spectra coverage to Jones et al. [2002]. Instead of using colour to visualise bugs, they use the Tarantula metric to rank program statements likely to be buggy. For a fair comparison of their proposed approach, they adapted other approaches, namely the Intersection model, the Union model, and the Nearest Neighbour model [Renieres and Reiss, 2003], using statement-based spectra coverage. Jones et al. [2005] also compare their proposed approach with the Cause Transition approach [Cleve and Zeller, 2005]. They use the successful diagnosis of bugs, $SucDiag$ measure (refer to Subsection 4.3.2 in Chapter 4) to compare the effectiveness of bug localization performance using their approach with the other proposed approaches. Within the 1% of code to be examined by the programmer, they show that Tarantula metric is able to locate 13.93% of the bugs as compared to 0%, 1.83%, 0%, and 4.65% of the bugs using the Intersec-

tion model, the Union model, the Nearest Neighbour model, and the Cause Transition approaches respectively.

Wong et al. propose several spectra metrics based on pass and fail test cases [Wong et al., 2007]. In their study, they emphasise giving less weight to pass test cases if a particular statement is executed by many pass test cases. They do not adjust any weights on the fail test cases, as they regard these test cases as important to indicate the likelihood of a statement being buggy. They use their tool, χSuds [Telcordia Technologies, Inc., 1998] to instrument and perform test executions of the program. They propose three different metrics, which we name Wong1, Wong2, and Wong3.

For Wong1, the metric value of a statement solely depends on the number of fail test cases. For Wong2, the metric value of a statement depends on both the number of pass and of fail test cases. For Wong3, Wong et al. [2007] introduce a constant value (known as α in their study) of 0.001 to adjust the weight of pass test cases. As there are more pass test cases that execute the statement of a program than fail test cases, lesser weight is assigned to the pass test cases. These metrics are shown in Table 2.3.

Wong et al. [2007] use the Siemens Test Suite as their benchmark in comparing the effectiveness of bug localization between their approach and Tarantula [Jones and Harrold, 2005]. They empirically evaluate the Wong3 metric with different constant values (α value). By using the Wong3 metric with the constant value of 0.001, they found the best improvement of bug localization performance compared to the Tarantula metric. They do not compare other metrics proposed by Renieres et al. and Abreu et al. [Renieres and Reiss, 2003, Abreu et al., 2006].

Most recently, Wong et al. propose another metric [Wong et al., 2010], which is different from their previous study [Wong et al., 2007]. In their study, the weight is adjusted on pass and fail test cases [Wong et al., 2010]. They use similar statement-based spectra coverage [Wong et al., 2007]. The only difference is that they use different spectra metrics to evaluate the program statements. They also choose several constant values (α value) and empirically observe that the smallest constant value of 0.0001 gives the best improvement of bug localization performance as compared to using the Tarantula metric. We name this metric as Wong4; the formula can be found as Definition 2.

Xie et al. recently propose to perform post-ranking of the program statements which have been evaluated with several spectra metrics [Xie et al., 2010]. They use the input of statement-based spectra coverage [Jones and Harrold, 2005, Wong et al., 2007, Wong et al., 2010]. Xie et al. [2010] use spectra metrics found in Naish et al. [2011] to generate a ranking of the program statements that are likely to be buggy. From the ranked program statements, they propose to group statements into two groups, and apply different heuristics to evaluate these groups of statements. We name this as a grouping-heuristic

approach. They propose to group ranked statements into a suspicious group, G_S and a non-suspicious group, G_U. If any statement has been executed by any fail test case ($a_{ef} > 0$), it is grouped as G_S. Otherwise, the statement is grouped as G_U. They propose heuristics for these groups. An additional metric value ($MetValue$) of 1 is added to the statements' $MetValue$ in G_S. Statements' $MetValue$ in group G_U are penalised by assigning a small metric value, which is the minimum metric value found in the group G_S.

Xie et al. [2010] evaluate on the single bug programs of the Siemens Test Suite. They investigate the improvement of bug localization performance using the existing approaches [Abreu et al., 2006, Naish et al., 2011] and their proposed grouping-heuristics approach. They observe that the Wong2, Wong3, Scott, and M2 metrics (known as M metric in their study) show improved bug localization performance after applying their proposed grouping-heuristic approach. They observe the O^p metric (optimal metric of single bug programs [Naish et al., 2011]) shows improvement of bug localization performance using their proposed grouping-heuristic approach on the $schedule$ program of the Siemens Test Suite. The latter observation is due to the inclusion of several programs in their study in which the buggy statement is not executed by any fail test case, due to *runtime error* (segmentation fault). As a result, the `gcov` tool fails to capture the coverage of the buggy statement. Therefore, the buggy statement for each of these programs has an a_{ef} of 0 and belongs to the non-suspicious group, G_U.

Debroy et al. proposed another type of post-ranking of program statements that have been evaluated with several spectra metrics [Debroy et al., 2010]. The basic idea of their proposed approach is to group statements based on the number of fail test cases executed for statements; that is, the a_{ef} (referred to as f in their study). Initially, the program statements are evaluated with spectra metrics, similarly to the previous studies [Jones and Harrold, 2005, Wong et al., 2007]. We define the pseudocode in Algorithm 1. The inputs to this algorithm are the ranked statements of program (based on the metric values of statements which have been evaluated with spectra metrics), along with the statement-based spectra coverage. This is followed by grouping statements based on the same number of fail test cases into groups of G_f, where f refers to f fail test cases. The groups G_f are sorted by the descending order of f; the group of statement(s) with the most fail test cases is sorted at the top. Finally, in each group of G_f, the statements are sorted again based on the metric values, and incorporated into a final ranking. Statements at the top of the ranking list are those most likely to be buggy.

Debroy et al. [2010] use χSuds [Telcordia Technologies, Inc., 1998] to perform program instrumentation and test suite execution, and gather statement-based spectra coverage. They evaluate their approach on single bug programs of the Siemens Test Suite, *grep*, and *gzip* datasets, using the Tarantula metric [Jones and Harrold, 2005] and the Ra-

Algorithm 1: Algorithm of Grouping-Based Strategy [Debroy et al., 2010]

Input: ranked statements of program evaluated with respective spectra metrics, statement-based spectra coverage
Output: ranked statements of program that are likely to be buggy

```
Group statements (in descending order) based on number of fail test cases
foreach statement s do
    if statement s is executed in f fail test cases then
        Group the statement s as part of group G_f;
    end
end
Sort group G_f according to f fail test cases (in descending order);
Sort again the statements based on their metric values within the sorted group G_f;
```

dial Basis function [Wong et al., 2008]. They handle ties, that is, statements with similar $MetValue$ to the buggy statement, using *High* and *Low* measures (the details of these measures can be found in Section 4.2 of Chapter 4). In their study, these measures are known as *Best case* and *Worst case* measures. They report their evaluation result using *High* and *Low* measures for different percentages of the program code to be examined by the programmer. For 10% of the program code to be examined by the programmer, they observe that their proposed grouping strategy is more effective than not using grouping strategy for the Siemens Test Suite, *grep*, and *gzip* programs respectively.

By using the proposed grouping strategy of Debroy et al. [2010] on the 10% of the program code to be examined by the programmer, the Tarantula metric (*High* measure) is able to locate 80% of the bugs in the Siemens Test Suite. Without using the grouping strategy, the Tarantula metric is only able to locate 55% of the bugs in the Siemens Test Suite. For the *grep* program, the Tarantula metric with the *High* measure is able to locate 90% of bugs as compared to 60% of bugs, using the grouping strategy and without using the grouping strategy respectively. On the *gzip* program, the Tarantula metric with the *High* measure is able to locate 95% of bugs as compared to 78% of bugs, using the grouping strategy and without using the grouping strategy respectively. Debroy et al. [2010] also show the same improvement on bug localization performance using their grouping approach on these datasets with the Radial Basis function [Wong et al., 2008]. The figures for other percentages of program code to be examined by the programmer using their proposed approach are detailed in their study [Debroy et al., 2010].

Renieres et al. propose the use of *difference spectra* (the differences between test cases) and *distance spectra* (using the Nearest Neighbour model) in order to locate bugs effectively [Renieres and Reiss, 2003]. They implement their proposed approaches in a tool called WHITHER. The pseudocode of their proposed approach can be referred to in Algorithm 2, where the input is the block-based spectra coverage. In their study, they use one fail test case and all the pass test cases.

Algorithm 2: Algorithm of Difference Spectra and Distance Spectra [Renieres and Reiss, 2003]

Input: block-based spectra coverage of a fail test case and all pass test cases
Output: Set of blocks of program likely to be buggy
Applying Different Pass and Fail Selection Strategies;
Intersection Model
$\cap S_p$(similar blocks of program executed by ALL pass test cases) $- S_f$;
Union Model
$S_f - \cup S_p$(any blocks of program executed by the pass test cases);
Nearest Neighbour Model
foreach *pass test case* **do**
    Coverage Type: Apply Hamming distance on the binary execution counts of S_p and S_f;
    Permutation Type: Sort blocks of program in S_p and S_f based on frequency execution counts before apply Hamming distance on both test cases;
end
Choose the pair of pass and fail test case with the least Hamming distance;
Set of blocks of program likely to be buggy is returned;

Difference spectra consists of two different models, namely the *Intersection model* and the *Union model*. The *Intersection model* refers to the differences in the set of program blocks executed in all the pass test cases, S_p, and the set of program blocks executed in the selected fail test case, S_f. The *Union model* refers to the differences between the set of program blocks executed in the selected fail test case, S_f, and the union of all the program blocks executed in all the pass test cases, S_p. The output of these models is the set of program blocks that are likely to be buggy.

Instead of using all the pass test cases to help locate the bug for *distance spectra*, only one of the pass test cases is selected for the *Nearest Neighbour* model. Using this model, only the pass test case that is most similar to the selected fail test case is chosen. In order to do that, Renieres et al. [2003] introduce two strategies, namely *Coverage Type* and *Permutation Type*. The former applies the Hamming distance [Hamming, 1950] on the binary execution counts of block-based spectra coverage of the pass test cases, S_p, and fail test case, S_f. These binary counts refer to whether each block of the program has been executed (1) or not (0). For the *Permutation Type*, Renieres et al. [2003] use the frequency counts of block-based spectra coverage. The frequency count considers the number of times each block of the program has been executed by each test case. They sort the program blocks by S_p and S_f before applying the Hamming distance. Finally, the pair of pass and fail test cases with the least Hamming distance is chosen. The set of the program blocks executed in the chosen pass test case, S_p, is subtracted from the set of the program blocks executed in the fail test case, S_f. The set of program blocks likely to be buggy is returned to the programmer.

From the set of program blocks produced by each model, Renieres et al. [2003] map the statements of the blocks using a program dependence graph (PDG) [Horwitz and Reps, 1992]. The program dependence graph represents the dependency and relationship

of the nodes of the program, which are represented with directed edges. Definition 12 in Subsection 4.3.3 is used to measure the effectiveness of their approach in bug localization performance. This measure refers to the amount of the program nodes (blocks) not needed to be examined in the program code (PDG) in order to locate the bugs. We defer the details of this performance measure to Subsection 4.3.3 in Chapter 4.

Renieres et al. [2003] evaluate 109 single bug programs from the Siemens Test Suite. They report that their proposed *Nearest Neighbour using Permutation Type* approach is able to locate more bugs than the *Intersection model*, *Union model*, and *Nearest Neighbour using Coverage Type* approaches. When not needing to examine 90% of the program nodes (that is, blocks of program) in the PDG, their proposed *Nearest Neighbour using Permutation Type* approach is able to locate 18 bugs in the test suite as compared to 1, 6, and 5 bugs using the *Intersection model*, *Union model*, and *Nearest Neighbour using Coverage Type* approaches respectively.

Abreu et al. propose to use block-based spectra coverage in order to locate program bugs [Abreu et al., 2006]. Their study is mainly motivated by Reps et al. [1997] and Abreu et al. [2006] are particularly interested to capture the block-based spectra coverage (referred to as block hit spectra). In their study, if a particular block of the program is executed, it increments the *block hit spectra count*. A comparison of the similarity of the blocks is made by applying different spectra metrics. They propose to use the Ochiai and Jaccard metrics (refer to the metrics in Table 2.3) in their evaluation.

In their study, Abreu et al. [2006] use diagnosis quality, q_d, to evaluate bug localization performance of their proposed approach. The diagnosis quality, q_d, is very similar to the rank percentages (see Definition 11). The block with the largest $MetValue$ indicates the block most likely to be buggy. If one or more blocks share similar metric values to the buggy block metric value, they use the *Low* measure. This measure is detailed in Section 4.2 of Chapter 4.

Abreu et al. [2006] evaluate using 118 (out of 132) programs of the Siemens Test Suite. Using diagnosis quality, q_d, they observe that the Ochiai and Jaccard to be more effective in locating bugs than Tarantula. They also observe that the Ample metric shows the worst bug localization performance.

Dallmeier et al. propose a development plugin known as **A**nalysing **M**ethod **P**atterns to **L**ocate **E**rrors (AMPLE) for Java IDE Eclipse [Dallmeier et al., 2005]. They use the **B**yte **C**ode **E**ngineering **L**ibrary (BCEL) [Dahm et al., 2002] for byte-code instrumentation of the program. In their study, program functions or methods belonging to the respective classes of the program are better known as method sequences. The pseudocode of their proposed approach is given in Algorithm 3. The inputs to this algorithm are the instrumented functions (methods) of program code and test cases (one pass test case and

Algorithm 3: Algorithm of Lightweight Bug Localization [Dallmeier et al., 2005]

Input: instrumented methods/functions of program code, a pass test case and all fail test cases
Output: classes of the program code ranked according to the weights of method sequences

```
foreach pair of a pass and fail test case not selected before do
    Execute the pair of test cases on the program code;
    Gather the coverage of respective method sequences invoked by object of classes of
    program for the pair of pass and fail test case;
    Identify the originating class that calls the respective method sequences;
    Compare differences of method sequences of the respective classes for the pair of pass and
    fail test case;
    More weight is assigned and aggregated to the originating class of the method sequences
    that appear in fail test case but not in pass test case;
end
Rank all classes of the program code according to the aggregated weights of method sequences;
```

all fail test cases). Initially, a pair of pass and fail test cases is executed to gather coverage of the sequences of methods invoked by objects of the respective classes of the program. This is followed by identifying the originating classes of the method sequences. A comparison is made between the method sequence(s) of pass and fail test cases for each class in the program. Originating classes of method sequences that only occur in fail test cases are assigned more weight. This step is repeated for all the pairs of pass and fail test cases. The weights of the originating classes of each method sequence are aggregated. Finally, each class of the program is ranked based on the aggregated weights of the method sequences. The program class with the highest aggregated weights of method sequences is the class most likely to hold the bug in the program code. In their plugin tool, Dallmeier et al. [2005] also propose the flexibility to adjust the number of methods in a method sequence if the programmer cannot locate the bug. They evaluate their proposed approach using the AspectJ compiler as their test subject. They observe that using seven methods in a method sequence, the programmer is able to locate the buggy class by inspecting only 1.98 classes of the program code. The study of Dallmeier et al. [2005] is later generalised by Abreu et al. [2006] and is known as the Ample metric (refer to the metric in Table 2.3) [Abreu et al., 2006, Abreu et al., 2007].

Hsu et al. propose a tool to rank more than one statement at a time; this tool is called **R**anking, **A**nalysis, and **P**attern **I**dentification for **D**ebugging (RAPID) [Hsu et al., 2008]. RAPID is used to instrument branches of the program. The pseudocode of this approach is presented in Algorithm 4. The inputs to this algorithm are instrumented program branches and branch-based spectra coverage. By using the branch-based spectra coverage, each instrumented branch of the program that has been executed by the test cases is mapped to the program spectra properties a_{ef}, a_{ep}, a_{nf}, and a_{np}. These branches are then evaluated using the Tarantula metric (refer to the metric in Table 2.3). The metric value, $MetValue$, is assigned to each branch of the program. Hsu et al. [2008] also gather the branch-based

Algorithm 4: Algorithm of RAPID [Hsu et al., 2008]

Input: instrumented branches of program, branch-based spectra coverage
Output: Sequential branches that are most likely to be buggy

1 Evaluate instrumented branch of program executed by the test cases using Tarantula metric;
2 Each branch executed in the program assigned $MetValue$ from Tarantula metric;
3 Gather branch-based spectra coverage of all fail test cases;
4 Map any branch $MetValue$ to the matching program branch executed in branch-based spectra coverage of fail test cases;
5 **Identifying sequences of buggy branches**;
6 **foreach** *branch-based spectra coverage of fail test case* **do**
7 | **if** $MetValue$ *of all the mapped program branches in the test case* $<$ *0.6 (threshold value)* **then**
8 | | Eliminate the fail test case;
9 | **end**
10 | **else**
11 | | Keep the branch-based spectra coverage of the fail test case in a new list, L;
12 | **end**
13 **end**
14 Identify common patterns of the remaining branch-based spectra coverage in L using BI-Directional Extension (BIDE) [Wang and Han, 2004];
15 Present the longest sequential branches (consist of statements) as the most likely branch to be buggy;
16 **if** *the programmer locates and understands the bugs from the sequential branches* **then**
17 | Exit;
18 **end**
19 **else**
20 | Append the previously presented branches to the next longest substring(s) of sequential branches found in list L;
21 | Repeat Steps 16–22 until bug is found by the programmer;
22 **end**

spectra coverage for all the fail test cases. In the subsequent step, they map each branch of the program (which has been assigned with metric value, $MetValue$) to the matching branch that has been executed in the branch-based spectra coverage of fail test cases. A typical fail test case can have more than one branch mapped with the assigned $MetValue$. They introduce a threshold value of 0.6 to remove any insignificant fail test cases. If all of the mapped branches of a particular fail test case have $MetValue$ of less than 0.6, the fail test case is not considered. Otherwise, the fail test case is mapped to a new list, L. The fail test cases, mapped in the list L, are used to identify common branch patterns using a data mining approach (BI-Directional Extension (BIDE)) [Wang and Han, 2004]. Eventually, the longest substring(s) of sequential branches that fulfil the threshold value of 0.6 and occur in all the fail test cases are chosen.

The following step is performed in an interactive mode. The sequential branches (usually consist of statements) are presented to the programmer as the most likely branches to be buggy. If the programmer is able to recognise the bug from the sequential branches, the algorithm stops. If the programmer still cannot understand and locate the bugs, the previ-

ously presented sequential branches are appended to the next longest sequential branches in the list, L. The latter branches are presented to the programmer. This step is repeated until the bug has been found by the programmer. Alternatively, Hsu et al. [2008] also suggest to lower the threshold value of 0.6 if the programmer still cannot locate the bugs from the presented sequential branches. They evaluate the Siemens Test Suite and indicate the effectiveness of their proposed approach by examining one of the examples in the Siemens Test Suite, which is *replace* program.

Santelices et al. [2009] investigate three different types of spectra coverage, namely statement-based spectra coverage, branch-based spectra coverage, and du-pair spectra coverage (those having data dependencies) [Offutt et al., 1996] with respect to the effectiveness of bug localization performance. Du-pair refers to associating the definition of a variable in a statement with the use of the same variable in the subsequent statement(s). An example of du-pair from Table 2.4 is Statement 2 and Statement 13 with variable *m*. Statement 2 defines the variable *m*, and it is subsequently used in Statement 13. Santelices et al. [2009] use a tool known as DUA-FORENSICS [Santelices and Harrold, 2007] to instrument Java programs. Some of the test programs in C are converted to Java before instrumentation is performed using DUA-FORENSICS.

Santelices et al. [2009] use Ochiai metric [Abreu et al., 2006] to rank the statements, branches, and du-pairs. For branches and du-pairs, they use similar approach as Hsu et al. [2008]. Initially, program code is instrumented with respect to statements, branches, and du-pairs. The test suite is executed over the instrumented program code to gather spectra coverage. In order to make a fair comparison with the statement-based spectra coverage approach, they use several rules to map the branches and du-pairs to statements. In their evaluation, they report the best bug localization performance among the different spectra coverage (statement-based, branch-based, and du-pairs). In their study, Santelices et al. [2009] show that overall, bug localization performance is the best by using du-pairs spectra coverage.

The disadvantage of using du-pairs spectra coverage is the time to instrument du-pairs as compared to statement-based and branch-based spectra coverage. Therefore, Santelices et al. [2009] also propose a technique to infer du-pairs spectra coverage from branch-based spectra coverage. More details of the latter technique can be referred to Santelices et al. [2007]. Santelices et al. [2009] investigate whether using the inferred du-pairs spectra coverage (which has less instrumentation overhead) shows any better bug localization performance than using statement-based and branch-based spectra coverage. In their evaluation, they show that the inferred du-pairs spectra coverage shows better bug localization performance than using the branch-based spectra coverage.

The Pinpoint framework has been proposed to determine and locate faulty components in large Internet services in the J2EE platform [Chen et al., 2002]. Dynamic analysis is used to gather web client request traces. Information of components that caused any failures (namely, caught exceptions) in the client requests are gathered. This information takes into account components often used and not often used in client requests. The Jaccard metric (refer to the metric in Table 2.3) has been used in the framework to relate the similarity of these components with failures.

The Zoltar metric (refer to the metric in Table 2.3) has been developed in the Embedded Software Lab of the Delft University of Technology as a modification of the Jaccard metric [A.Gonzalez, 2007]. A new term that includes a constant of 10 000 is introduced as the denominator of this metric to distinguish non-buggy and buggy blocks of program. In their study, the metric has been used to compare bug localization performance with Tarantula, Ochiai, and Jaccard metrics.

There is another bug localization domain that locates bugs based on predicates [Liblit et al., 2005, Liu et al., 2005]. We refer to it as predicate-based spectra coverage. A predicate is defined as a part of program properties; for example, branches, return values, and scalar pairs. An example of branches would be conditional statements such as if-then-else. The `return` statement is used to return control to the calling function. Example of return predicates are `return` values of greater than zero, less than zero, and equal to zero. Scalar pairs refers to the boundary when a variable assignment occurs. For example, whether the left side of the assignment is less than/greater than/equal to the right side of the assignment.

Liblit et al. develop a system known as Co-operative Bug Isolation (CBI) [Liblit et al., 2005]. The system gathers information of the user executions automatically when the program crashes. Similar systems have been deployed by Microsoft, GNOME, and KDE in order to gather the automated crash reporting information from the user executions. Initially, program code is instrumented with respect to selected predicates, namely branches, return values, or scalar pairs. Instead of gathering information of predicates of the program for every user execution, a representation of the predicates of the program from the user executions are gathered. Liblit et al. perform sparse random sampling on the respective predicates using geometrically distributed random numbers [Liblit et al., 2003]. The details of the sampling approach can be referred to in Algorithm 13 in page 45. These predicates consist of counter variables to determine the next predicates to be sampled. In their study, each predicate $Pred$ is assigned with *F(Pred)*, *S(Pred)*, *F(Pred observed)*, and *S(Pred observed)* respectively. *F(Pred)* refers to the number of fail tests that the $Pred$ was executed and *True*. *S(Pred)* refers to the number of pass tests that the $Pred$ was executed and *True*. *F(Pred observed)* refers to the number of fail tests where the $Pred$ was

Table 2.5: Spectra Metrics used in the Predicate-based Spectra Coverage studies

Name	Formula
Failure(Pred)	$\frac{F(Pred)}{S(Pred)+F(Pred)}$
Context(Pred)	$\frac{F(\textit{Pred observed})}{S(\textit{Pred observed})+F(\textit{Pred observed})}$
CBI Inc(Pred)	Failure(Pred)-Context(Pred)
CBI Log(Pred)	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\log totF}{\log F(Pred)}}$
CBI Sqrt(Pred)	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\sqrt{totF}}{\sqrt{F(Pred)}}}$
FPC(Pred)	Failure(Pred)+Context(Pred)
OFPC(Pred)	3*F(Pred)+FPC(Pred)
O8FPC(Pred)	10*F(Pred)+8*Failure(Pred)+Context(Pred)

observed (reached). *S(Pred observed)* refers to the number of pass tests where the $Pred$ was *observed* (reached). The term of *observed* is also known as *reach*. We showed a simple example of if-then-else to demonstrate the differences between *True* and *reach*. The following excerpt in Figure 2.1 shows that if the predicate on Statement 1 (S1) is *True*, it will cause the program to crash. The predicate on Statement 1 (S1) and its negation (S4) is *reach* if and only if S1 is executed. The negation of the predicate on S1 is *True* if and only if Statement 4 (S4) is executed.

```
S1:     if (f == NULL){
S2:             x=0;
S3:             *f;
S4:     }else    x=1;
```

Figure 2.1: Excerpt of if-then-else Predicate

Using the notation of *F(Pred)*, *S(Pred)*, *F(Pred observed)*, and *S(Pred observed)*, Liblit et al. [2005] compute *Failure(Pred)* and *Context(Pred)*. *Failure(Pred)* is defined as the number of fail tests where $Pred$ was *True* divided by the total number of tests where $Pred$ was *True*. *Context(Pred)* is defined as the number of fail tests where $Pred$ was *observed* (reached) divided by the total number of tests where $Pred$ was *observed* (reached). These metrics are summarised in Table 2.5.

A brief pseudocode is defined in Algorithm 5. The input is the instrumented predicates either in branches, return values, or scalar-pairs schemes. Another input of this algorithm is the user execution information, represented as feedback report, R. This report indicates whether the program is Pass or Fail. It also contains the information of whether each $Pred$ is *observed True* or otherwise. Liblit et al. [2005] perform sparse random sampling [Liblit et al., 2003], using geometrically distributed random numbers, and use predicate

Algorithm 5: Algorithm of Statistical Bug Isolation [Liblit et al., 2005]

Input: instrumented predicates (if-then-else, return values, or scalar-pairs scheme), user executions represented as feedback report R

Output: Predicates ranked according to Importance(Pred) based on *F(Pred)*, *Increase (Pred)*, and *CBI Log (Pred)*

1 Perform sparse random sampling approach [Liblit et al., 2003] on instrumented predicates which are contained in R and gather the predicate-based spectra coverage for the sampled predicates;
2 Compute $Failure$ and $Context$ for each instrumented predicate, $Pred$;
3 Rank the *Importance* of all the $Pred$ based on *F*, *CBI Inc*, and *CBI Log* (see the metrics in Table 2.5);

counters to record how often each predicate, $Pred$, is executed in each user execution. This information forms the predicate-based spectra coverage. $Failure$ and $Context$ are computed for the respective sampled $Pred$. Liblit et al. [2005] propose several metrics to rank the predicates of the program that are likely to be buggy. One of them is $Increase$ (later renamed as *CBI Inc*). There are also two variations, which attempt to combine $Increase$ with another simple metric, namely the proportion of fail tests in which $Pred$ is *True* using the harmonic mean. One variation uses logarithms [Liblit et al., 2005]; we name this variation as *CBI Log*. The other variation uses square roots; we name this *CBI Sqrt*. These metrics can be found in Table 2.5. In their study, F, *CBI Inc*, and the *CBI Log* harmonic mean variations are used to rank the likelihood of each predicate, $Pred$, to be buggy. Liblit et al. [2005] refer to this likelihood as $Importance$. For all these metrics in Table 2.5, we use a similar notation to their study except with the notation $Pred$ (termed P in their study).

Liblit et al. [2005] evaluate their proposed approach on the MOSS, CCRYPT, BC, EXIF, and RHYTHMBOX datasets [Rhythm, nd]. By using their proposed predicate-based metrics, they analyse the predicates that are ranked top on these datasets. They show that these predicates are the bugs in the programs. They do not make any comparison with the Tarantula metric [Jones and Harrold, 2005] or with other metrics previously proposed by Renieres et al. [2003].

Chilimbi et al. recently use path-based spectra coverage to improve bug localization performance [Chilimbi et al., 2009]. They develop a tool known as HOLMES and propose two debugging approaches known as Non-adaptive and Adaptive debugging.

Algorithm 6: Algorithm of Holmes - Non-adaptive debugging [Chilimbi et al., 2009]

Input: instrumented program code (path-based) $Path$, path-based spectra coverage

Output: paths ranked according to *Importance* of $Path$ (likely to be buggy) based on *Failure*, *CBI Inc*, and *Sensitivity*

1 $Failure$ and $Context$ are computed for each instrumented path of the program code, $Path$;
2 Rank the *Importance* based on *Failure*, *CBI Inc*, and *Sensitivity* of all the $Path$;

In the non-adaptive debugging approach, Chilimbi et al. [2009] use framework similar to the CBI system [Liblit et al., 2005], by gathering user execution information represented in reports, R. From the reports, they gather the path-based spectra coverage for each path of the program. A brief pseudocode is described in Algorithm 6 (Non-adaptive debugging). The inputs to this algorithm are the path-based instrumented program $Path$, as well as the path-based spectra coverage of each path of the program. Chilimbi et al. [2009] adapt the *Failure* and *Increase* metrics [Liblit et al., 2005] in order to rank the paths most likely to be buggy. *Sensitivity* is used in their study to rank the importance of the paths likely to be buggy. It refers to the proportion of each path being executed by fail test cases with respect to the number of fail test cases in the program.

Algorithm 7: Algorithm of Holmes - Adaptive debugging [Chilimbi et al., 2009]

Input: program code, user executions information of a program
Output: Paths likely to be buggy returned to the programmer
1 Monitor each program execution for failures and gather reports;
2 Perform static analysis based on the reports;
3 Compute set of functions appear in fail stack traces (most likely root cause of the program failure);
4 Instrumented program code especially on the selected set of functions;
5 Monitor several program execution for failures, and gather reports and profiles for the selected set of functions;
6 Use static analysis component again to analyse and model sets of buggy functions with *Importance*;
7 **while** *strong bug predictors not obtained or all failures not explainable to the programmer* **do**
8 | Invoke the static analysis component again to identify other fragments of code (weak bug predictors), which are related to the previous predictors;
9 **end**
10 Bug is found and stop;

In order to gather path-based spectra coverage for the non-adaptive approach (Algorithm 6), this method needs to gather user execution information before sparse random sampling of program paths can be performed on the paths of the program. This causes some instrumentation overhead (similar to that of the study in Liblit et al. [2005]) to maintain the path counters and sample the paths of the program. Therefore, Chilimbi et al. [2009] propose an adaptive debugging approach in Algorithm 7 to locate bugs without compromising the overhead of instrumentation. The inputs to this algorithm are the program code and user executions information of a program. Initially, they monitor several user executions that cause the program to fail, and gather the same reports used in Liblit et al. [2005]. These reports are generated automatically whenever the program crashes or fails. Once they have sufficient reports, they use the static analysis component embedded in the HOLMES tool to analyse the bug reports. In this step, they extract set of program code functions that appear in the fail stack traces from the reports. These functions ulti-

mately form paths likely to be buggy. This is followed by instrumenting the program code particularly on the set of functions extracted previously. The program code is executed by all the user executions again, except that only the coverage information of the extracted set of functions is gathered. This information is analysed again using the tool's static analysis component. The set of functions likely to be buggy are computed using *Importance* based on *Failure*, *CBI Inc*, and *Sensitivity* for each $Path$ of the program. Finally, functions that are ranked highest (known as *strong bug predictors*) are presented to the programmer. If they are explainable as the bug to the programmer, the algorithm stops immediately. Otherwise, the static analysis component is invoked to identify new sets of functions in the fail set traces which are likely to cause the program to fail. These sets of functions are known as *weak bug predictors* in their study as they are not chosen in the first iteration. Steps 7–9 of Algorithm 7 are repeated continuously until the bug is found and explainable to the programmer. Chilimbi et al. [2009] evaluate their proposed approach on 6 out of 7 programs of the Siemens Test Suite [Do et al., 2005] using the Microsoft Phoenix compiler [Hall et al., 2009]. They also evaluate on other programs such as *gcc*, *translate*, and *edg*.

Chilimbi et al. [2009] evaluate their performance using the program dependence graph (PDG), similar to the previous studies [Renieres and Reiss, 2003, Cleve and Zeller, 2005]. The details of this performance measure are given in Subsection 4.3.3 of Chapter 4. By examining 10% of the program nodes in the PDG of the Siemens Test Suite, 24, 14, and 0 bugs can be located using the path-based, predicate-based, and branch-based spectra coverage respectively. Therefore, they conclude that path-based spectra coverage performs best in locating bugs as compared to using the predicate-based and branch-based spectra coverage.

Recently, we have investigated the relationship of statement-based spectra coverage and predicate-based spectra coverage [Naish et al., 2010]. We propose several heuristics to reconstruct predicate-based spectra coverage (if-then-else) using statement-based spectra coverage. We observe that by using the predicate-based spectra coverage gives additional information that helps improve bug localization performance as compared to using the statement-based spectra coverage. Initially, we determine whether each statement is at the start of a *then* block or at the end of an *else* block of the program. Once the statements have been identified to be in the *then* or *else* block, statements not in the *then* or *else* block are identified next. The statement-based spectra coverage is propagated forward and backward for the *then* and *else* block.

We evaluate our proposed approach on a model program which has more statements than the model program $ITE2_8$ (the details of this model program are given in Chapter 5). The predicate-based spectra coverage are reconstructed from the statement-based spectra

coverage of the model program. We evaluate on several spectra metrics, including the CBI Log and CBI Sqrt originally used for statement-based spectra coverage [Naish et al., 2011]. We also propose several new spectra metrics, namely FPC, OFPC, and O8FPC, for predicate-based spectra coverage (these metrics are listed in Table 2.5). The FPC metric refers to the addition of the Failure and Context metrics. The OFPC and O8FPC metrics are variants of OFPC based on an understanding of the single bug optimal metric, O^p, proposed by Naish et al. [2011]. We conclude by using the model program on several number of tests, the O8FPC metric outperforms other metrics, including the CBI Inc, CBI Log, and CBI Sqrt metrics [Liblit et al., 2005]. We also use these metrics to evaluate the predicate-based spectra coverage of the Siemens Test Suite [Do et al., 2005], the subset of Unix Test Suite [Wong et al., 1998], and the Concordance [Ali et al., 2009]. We observe that the O8FPC metric has the average rank percentages of 9.95% as compared to 10.11% using O^p metric in single bug Concordance programs. Other predicate-based metrics of CBI Inc, CBI Log, and CBI Sqrt yield average rank percentages of 50.04%, 49.05%, and 51.18% respectively on the single bug Concordance programs. We also observe that the O8FPC performs better than other spectra metrics, which were originally used in statement-based spectra coverage.

Statistical model-based bug localization (better known as SOBER) [Liu et al., 2005] also uses the predicate-based spectra coverage. In their study, Liu et al. [2005] instrumented predicates with branches and returns. The difference between their study and Liblit et al. [2005] is that the information of execution counts (frequency counts) is used instead of binary execution information in the CBI system [Liblit et al., 2005]. SOBER captures the number of times a particular predicate, $Pred$, has been executed in pass and fail test cases respectively. Another difference of SOBER and the CBI system [Liblit et al., 2005] is that the former does not rely on the user execution information and sparse random sampling proposed by Liblit et al. [2003]. Liu et al. [2005] use the existing test cases of the benchmarks to instrument and gather the predicate-based spectra coverage.

In SOBER, Liu et al. [2005] define the probability of $Pred$ which has been executed and *True* for each test case. This is referred to as the *evaluation bias* of $Pred$, $E(Pred)$. They model the distribution of the probability of $E(Pred)$ with respect to pass and fail test cases, which is $E(Pred)_{pass}$ and $E(Pred)_{fail}$ respectively. Finally, they consider the differences of $E(Pred)_{pass}$ and $E(Pred)_{fail}$ as to indicate the likelihood of predicate, $Pred$, of the program being buggy. The higher the difference for a predicate, $Pred$, the more likely the predicate, $Pred$, is to be buggy.

Liu et al. [2005] also use statistical measures, such as the mean and variance to develop a ranking score, $s(Pred)$, to rank the predicate, $Pred$, accurately. A predicate conformity, $p(Z)$, is measured according to the standard normal distribution (see Definition 5). They

evaluate their approach on the Siemens Test Suite and the BC1.06 datasets. Their study also reports the performance measure of their proposed approach using the program dependence graph (PDG). They use the complement of the performance measure proposed by Renieres et al. [2003] (see Definition 12 in page 78). This refers to the number of program nodes needed to be examined in the program code (PDG) in order to locate bugs. We defer the details of this performance measure to Subsection 4.3.3 in Chapter 4.

Definition 5. *Ranking score, s(Pred)*

$$s(Pred) = log(\frac{\sigma_{Pred}}{\sqrt{\mu}p(Z)})$$

Liu et al. [2005] observe that within the 10% of the program code being examined by the programmer, they are able to locate 68, 52, and 34 bugs in the 130 programs of the Siemens Test Suite, using their proposed approach of SOBER, the proposed approaches of Liblit et al., and Cleve et al. [Liu et al., 2005, Liblit et al., 2005, Cleve and Zeller, 2005] respectively. They evaluate their proposed approach on BC1.06 and observe that the predicates ranked most likely to be buggy are very close to the bug that causes the program to crash.

The proposed approach of Liu et al. [2005] uses more information of predicates than that of Liblit et al. [2005]. They consider multiple evaluations of predicates (frequency counts) for each test case. Liblit et al. only assume each predicate is observed once, even if it is executed multiple times in each test case [Liblit et al., 2005]. Another advantage of SOBER is less overhead in the program instrumentation, since only branches and return values are instrumented as predicates.

2.5 Summary

In this chapter, we introduced the notion of program spectra and spectral debugging approach to locate bugs. We gave a comprehensive list of various spectra metrics used in the thesis. We also briefly detailed the origin of these spectra metrics. Several studies that introduced spectra metrics to locate bugs have been described in this chapter. We also surveyed several studies of spectral debugging which are closely related to the thesis.

3

Survey of Software Fault Localization Techniques

3.1 Introduction

In Chapter 2, we have briefly introduced several approaches based on software fault localization techniques which are closely related to the contributions of this thesis. In this chapter, we consolidate a survey of detailed studies on software debugging, particularly dynamic analysis. These studies show that there are several different approaches to bug localization, namely: slicing and dicing; statement-based, block-based, and predicate-based spectra coverage; state-based; test reduction; and combining spectra-based and machine learning approaches.

3.2 Slicing and Dicing Approaches

We discuss several studies that analyse the slicing approach [Lyle, 1985, Weiser and Lyle, 1986, Agrawal and Horgan, 1990]. This has been one of the earliest approaches proposed in the area of debugging. Slicing refers to identifying the parts of a program (a set of statements or program blocks) that affect the value of a particular program variable. There are two types of slicing approaches, namely static slice and dynamic slice. Static slice uses the information in a control flow graph (CFG) of a program to locate the parts of the program that affect the value of a particular variable within it [Weiser and Lyle, 1986]. Dynamic slice relies on the information of the test suites, which is the test coverage gathered through dynamic analysis to determine the parts of a program that affect the value of a particular program variable [Korel and Laski, 1990]. In this section, we refer to a dynamic slice as an execution slice.

We also discuss dicing approach and how it improves upon that of slicing approach.

Dicing refers to the parts of a program that appear in one slice but not in another slice [Lyle and Weiser, 1987]. Dicing can be further divided into static dice and dynamic dice. These approaches use similar information to that of static and dynamic slice to identify the parts of a program that affect the value of a particular program variable. Neither slicing nor dicing considers applying any spectra metrics to rank program statements that are likely to be buggy. Instead, these approaches narrow down the parts of a program (statements or program blocks) that are likely to be buggy in order to help the programmer debug the program code.

Chen et al. propose to narrow down the search of a bug in the dynamic slice approach [Chen and Cheung, 1993]. There are two types of dynamic slice, namely successful execution slice and failure execution slice. Different terminologies are used to refer to the types of dynamic slices in several studies [Chen and Cheung, 1993, Agrawal et al., 1995, Wong and Qi, 2004]. We adopt the terms used in Agrawal et al. [1995] – successful execution slice and failure execution slice – throughout this chapter. In their study, Chen et al. [1993] use test inputs of dynamic slices to determine the successful and failure execution slice. Multiple test inputs can have an identical dynamic slice. An execution slice is considered successful, if the output value of its slicing variable is correct when applied with all the associated set of test inputs of the dynamic slice. Otherwise, the execution slice is considered as a failure execution slice. They propose several strategies to construct a dynamic dice by using successful and failure execution slices of a program. They are listed as below:

1. Removing program statements in a failure execution slice which have been executed in a successful execution slice.

2. Removing program statements in a failure execution slice which have been executed at least once in the successful execution slices.

3. Removing program statements in a failure execution slice which have been executed in all of the successful execution slices.

4. Removing program statements which have been executed in all of the failure execution slices from the successful execution slices.

5. Removing program statements which have been executed in all of the successful and failure execution slices.

The first strategy involves a successful and a failure execution slice. This strategy retains at least one of the bugs in the dynamic dice. The first strategy is better than the second strategy if the bugs of the program are in different failure execution slices. Chen

et al. [1993] observe that the third strategy ensures less bugs are missed when compared to the first strategy. They observe that using the dynamic dice has more advantages than using the static dice [Lyle and Weiser, 1987].

Agrawal et al. investigate the effectiveness of using the dynamic dice approach to narrow down the search of a bug [Agrawal et al., 1995]. Their study is referred to as a dicing approach in several studies [Hao et al., 2005, Hao et al., 2006, Hao et al., 2008]. Agrawal et al. [1995] use base version program code as the *oracle* of a program in order to determine the correctness of its slices (successful and failure execution slices). In their study, a *successful execution slice* is the slice of the fault-seeded program code version for which the output of test executions on the slice of the program code is similar to the output of the test executions on the slice of the base version program code. A *failure execution slice* is the slice of the fault-seeded program code version for which the output of test executions on the slice of the program code is different from the output of the test executions on the slice of the base version program code. They use the χSlice [Telcordia Technologies, Inc., 1998] tool to perform the slicing and dicing of the program. The χSlice tool allows the programmer to visualise the execution slices and dices of a program. Pseudocode representing the search for a bug in the dynamic dice is described in Algorithm 8.

Algorithm 8: Algorithm of the Dynamic Dice Approach [Agrawal et al., 1995]

Input: successful execution slices, failure execution slices
Output: present the dices to the programmer to debug the program code
1 Form all possible dices by subtracting the respective successful execution slices from the failure execution slices;
2 Compute the number of statement(s) in the dice, *average size*;
3 Compute the number of buggy statement(s) in the dice, *good dices*;
4 Present one of the dices randomly to the programmer to debug the program code;

The inputs to this algorithm are the set of *successful execution slices* and *failure execution slices* of the program. Initially, these execution slices are obtained with respect to the block-based coverage type granularity using the χSlice tool [Telcordia Technologies, Inc., 1998]. This is followed by obtaining the dice for each pair of successful and failure execution slices. The maximum number of the dices that can be formed (pairwise combination) is the multiplication of the total number of the successful and failure execution slices. In order to measure the effectiveness of the dynamic dice approach, Agrawal et al. [1995] determine the *average size* and the *good dice* for each dice. The *average size* refers to the number of statements (including the bug) in each dice. The *good dice* refers to the number of the buggy statements in the dice. Finally, the programmer is presented with one of the dices randomly to debug the program code. They observe that the *good dice* is an important condition to locate the bugs. The GNU Unix Sort program is used to study the effectiveness of their proposed approach. This program consists of 914 lines of code

and 56 test cases. It contains 25 versions of the program with various single seeded bugs. The advantage of using this approach is that the programmer is able to focus on a smaller portion of the program code, its dices, to narrow down the search of its bugs.

Pan et al. build a debugging tool known as *Spyder* [Pan and Spafford, 1992] to determine the dynamic slices of a program. Their study relies on the *oracle* of the program to determine the correctness of the dynamic slices (the successful and failure execution slices). They introduce two metrics known as the *inclusion frequency* and the *influence frequency*. The *inclusion frequency* of a program statement s refers to the number of distinct dynamic slices that contain the statement s. The *influence frequency* of a program statement s refers to the number of times the statement s is executed in the dynamic slices. Based on these metrics, they introduce several heuristics to narrow down the statements that are likely to be buggy in the dynamic slices. One of the heuristics is choosing the statements that have been executed in all the failure execution slices of the program. The fine details of the other heuristics can be found in their study [Pan and Spafford, 1992]. For each of the heuristics, the *Spyder* tool produces a set of statements of the program suggested to be the bug from the dynamic slices. They evaluate the effectiveness of their proposed heuristic approaches by considering the ratio of the number of program statements that are related to the bugs with respect to the number of statements suggested to be the bug using the heuristics. A higher ratio for a particular heuristic indicates that it is more effective in narrowing down the search of the bugs. They apply their heuristics on 11 different programs. They conclude that using heuristics with the *inclusion frequency* metric is more effective in narrowing down the search of the bugs in these programs.

Korel et al. extend the dynamic slice approach [Korel, 1988] using a tool known as **P**rogramming **E**rror **L**ocating **A**ssistant **S**ystem (PELAS) [Korel and Rilling, 1997] to help the programmer understand program executions in the Pascal language. The tool is used to execute the program code (dynamic analysis) and to display the sections of program executions that belong to a dynamic slice. A list of all the related variables of the program likely to be buggy are presented to the programmer. They also propose the use of partial dynamic slice when the programmer chooses to analyse portions of a program, for example, the executions of specific loops and procedures.

Wong et al. propose an improved dynamic slice approach [Wong and Qi, 2004] to locate the bugs in a program. Their study incorporates the relationship of data dependencies between the blocks of a program to locate its bugs. By using their proposed approach, bugs that are in both successful and failure execution slices can be located. They propose two approaches, namely the *augmentation approach* and the *refining approach*. The *augmentation approach* is only used if the bug is not found in the dynamic dice. This is an iterative approach of including additional block(s) of a program to be examined in

the dynamic dice until the bug is found. The *refining approach* is an iterative approach to remove block(s) of the program that do(es) not contain the bug in the dynamic dice. Using the latter approach helps the programmer to examine lesser program code in the dice to locate the bugs. They develop a tool known as **D**ebugging Tool Based on **E**xecution **S**lice and **I**nterblock data **D**ependency (**DESiD**). For both *augmentation* and *refining* approaches, dice D^1 is initially defined by taking the difference of a failure execution slice, E_f, from the successful execution slice, E_s: that is, dice D^1 contains the sets of blocks of a program that are executed in the failure execution slice, E_f, but not in the successful execution slice, E_s.

Algorithm 9: Augmentation Approach Algorithm

Input: dice D^1 and failure execution slice, E_f
Output: code segment containing bug(s)

```
1  k=1;
2  Set θ: E_f-D^1;
3  Augment code segment, A^k : (β ∈ θ) ∧ (β △ D^1);
4  if bug is in code segment A^k then
5  |  Stop;
6  end
7  else
8  |  Set iteration, k++;
9  |  Augment code segment, A^k=A^{k-1} ∪ (β ∈ θ ∧ β △ A^{k-1});
10 end
11 if A^k is similar to A^{k-1} then
12 |  Reach final augmented code segment and no further code segment is needed to be
   |  constructed;
13 |  Stop and return E_f to the programmer to examine the bug;
14 end
15 else
16 |  Go to Step 4;
17 end
```

The pseudocode of the *augmentation approach* is described in Algorithm 9. This algorithm is executed if the bug is not found in D^1. We define several notations in this algorithm. θ is defined as the blocks of the program being debugged that only appear in E_f but not in D^1. The blocks of the program in θ are potentially non-buggy blocks. Initially, the code segment, A^k, is augmented by identifying a block β that is an element in θ and has a data dependent relationship (represented with $\triangle$) with D^1. An example of a data dependency relationship: if β uses variable x which has been defined in D^1. The code segment that has been augmented, A^k, is presented to the programmer. The programmer examines whether any bug is contained in the augmented code segment. If a bug is found in the augmented code segment, A^k, the algorithm stops immediately. Otherwise, the iteration of k is incremented. The next augmented code segment A^k will be based on the code segment A^{k-1}. In the augmented code segment, A^k, a block of the program, β that is

part of θ and has a data dependent relationship with A^{k-1} is identified. A^{k-1} is essentially a subset of A^k. The augmented code segment, A^k, is also checked against the previous augmented code segment, A^{k-1}, to determine if they are identical. The code cannot be augmented if they are identical and the programmer is then presented with the failure execution slice, E_f, to locate the bug. Otherwise, the *augmentation approach* is repeated until the bug is found or it reaches the final augmented code segment.

Pseudocode of the *refining approach* is presented in Algorithm 10. This approach assumes that the bug has been found in the existing dice, represented by D^1. The motivation of this approach is to refine and remove the additional code in the dice D^1 that must be examined by the programmer to locate the bug. The inputs to this algorithm are dice D^1, the successful execution slices, and one failure execution slice. Initially, k successful execution slices are selected. A new dice is constructed based on the existing dice D^1 (which contains the bug) and the union of the additional k successful execution slices. The dice is presented to the programmer to examine whether the bug is contained in the dice. If there is a bug in the refined dice D^{k+1}, the algorithm stops and return the refined dice to the programmer. If there is no bug in the dice, the k counter decrements and Steps 2–15 will be repeated to construct and refine a new dice. These steps are repeated unless the bug is found. If the minimum number of k successful execution slices has reached 0 (k==0), this indicates that the bug has not been found in any of the refined dices. Therefore, the programmer will use the existing dice D^1 to examine the bug.

Algorithm 10: Refining Approach Algorithm

```
   Input: dice D^1, successful execution slices and failure execution slice
   Output: Dice containing bug(s)
 1 Randomly select k successful execution slices;
 2 Construct new dice D^{k+1}:D^1-∪ (k^{th} successful execution slices);
 3 if bug is found in dice D^{k+1} then
 4 |   Stop and present dice to programmer;
 5 end
 6 else
 7 |   Set k − −;
 8 |   if k==0 then
 9 |   |   Stop;
10 |   |   Examine code in dice D^1;
11 |   end
12 |   else
13 |   |   Repeat Steps 2–15;
14 |   end
15 end
```

In their study, Wong et al. [2004] evaluate and show the robustness of their proposed approaches on the set of Space programs [Do et al., 2005] using the DESiD tool. The details of the Space programs are described in Section 4.5. Wong et al. [2004] observe

that the *augmentation approach* is effective in locating bugs, especially when the bugs do not exist in the dice. For the *refining approach*, they also observe that the programmer can examine less program code in order to locate the bugs. In practice, the programmer often does not know the exact location of the bugs in a program. They suggest using the *refining approach* before using the *augmentation approach* to locate the bugs.

3.3 Statement-based, Block-based, and Predicate-based Spectra Coverage Approaches

We discuss several studies using the statement-based, block-based, and predicate-based spectra coverage approaches in this section. The details of these approaches can be found in Chapter 2. These approaches instrument a program with respect to the statements, blocks, and predicates of the program to capture its respective execution coverage information. Test case information is used to rank the instrumented statements, blocks, or predicates of the program according to the likelihood of them being buggy.

Hao et al. propose a fuzzy set theory approach to help locate bugs (referred to as testing-based fault localization (TBFL) in their study) [Hao et al., 2008]. In a fuzzy set theory approach, each program statement is assigned a membership grade. This membership grade eventually determines the statements that are likely to be buggy. The fine details of the fuzzy set theory approach can be found in Zadeh [1965].

Algorithm 11: Algorithm of Bug Localization using Fuzzy Set Theory Approach [Hao et al., 2008]

Input: statement-based spectra coverage
Output: ranked program statements likely to be buggy
1 Find the membership grade of each test case;
2 **foreach** *statement s in program* **do**
3 | Find the set of test cases that execute statement s, $Appear_s$;
4 | Find the set of test cases that execute statement s and fail, $Fail_s$;
5 **end**
6 **foreach** *statement s in program* **do**
7 | Calculate A_s, the maximum of the membership grade for statement s in $Appear_s$;
8 | Calculate F_s, the maximum of the membership grade for statement s in $Fail_s$;
9 | Calculate suspiciousness of statement, s as $P(s) = \frac{|F_s|}{|A_s|}$;
10 **end**
11 Rank statements based on suspiciousness $P(s)$ in decreasing order;

The pseudocode of the proposed approach of Hao et al. [2008] is described in Algorithm 11. The input to this algorithm is the statement-based spectra coverage (binary form) shown in Table 2.2. Initially, a membership grade is computed for each test case to normalise the test coverage information. In the next step, conditional probability is ap-

plied to the program statements [Newmark, 1988]. For each statement s, a set of test cases that execute the statement s, $Appear_s$, is determined. A set of test cases that execute the statement s and fail, $Fail_s$, is also determined. For each statement s, the maximum membership grade of test t that executes the statements of the program for both $Appear_s$ and $Fail_s$ is computed. Finally, the suspiciousness value for the statement s is computed using the $P(s)$ function (Step 9 of Algorithm 11). Hao et al. [2008] evaluate their proposed approach on the Siemens Test Suite [Do et al., 2005], Tiny C Compiler (TCC) [Bellard, 2010], desk calculator (DC) [Morris and Cherry, 1983], and Counter of directory sizes (CNT) [CNT, 2010] benchmarks. They evaluate their proposed fuzzy set theory approach on the test suites of redundant test cases and non-redundant test cases (the details of non-redundant test cases can be found in Chapter 7).

For a fair comparison with other proposed approaches, Hao et al. [2008] report bug localization performance using the successful diagnosis of bugs, $SucDiag$ measure (see Subsection 4.3.2). They compare the effectiveness of their proposed approach with other approaches such as Nearest Neighbour [Renieres and Reiss, 2003], Cause Transition [Zeller, 2002], Tarantula [Jones and Harrold, 2005], SOBER [Liu et al., 2005], and CBI [Liblit, 2004]. Hao et al. [2008] observe that their proposed approach outperforms the Nearest Neighbour approach. By examining up to 1% of the program code in the Siemens Test Suite, they successfully locate 0%, 2.48%, and 13.93% of the bugs using the Nearest Neighbour, their proposed fuzzy set theory approach, and the Tarantula approach respectively. Hao et al. [2008] also compare their proposed approach with the Dicing [Agrawal et al., 1995] and Tarantula [Jones and Harrold, 2005] approaches. They observe that their proposed approach has similar effectiveness (in terms of bug localization performance) as the Dicing approach [Agrawal et al., 1995] regardless of the test suites used (redundant test cases and non-redundant test cases).

Hao et al. extend their previous fuzzy set theory approach using an interactive bug localization method [Hao et al., 2006]. The pseudocode of this method is shown in Algorithm 12. This algorithm is almost similar to the previous approach [Hao et al., 2008] except it provides extra interactions for the programmer to locate a bug. The input to this algorithm is the statement-based spectra coverage. Initially, the fuzzy set theory approach is applied to the statement-based spectra coverage to produce a list of statements most likely to be buggy using the suspicious function $P(s)$ [Hao et al., 2008]. Using this function, these statements are ranked in the descending order before presented to the programmer.

The highest ranked statement s in this list is initially set as the check point for the programmer to debug the program code. The programmer determines whether the check point (statement s) is the bug. If the check point is determined to be the bug by the pro-

Algorithm 12: Algorithm of Interactive Bug Localization using Fuzzy Set Theory Approach [Hao et al., 2006]

Input: statement-based spectra coverage
Output: buggy statement is found by the programmer, number of check points before the bug is found

1 Use fuzzy set theory approach and $P(s)$ [Hao et al., 2008] on the statement-based spectra coverage to generate ranked program statements;
2 Statement s ranked top based on the $P(s)$ is set as a check point;
3 **if** *the programmer determines the statement s is buggy* **then**
4 | Stop and bug localization task accomplished;
5 **end**
6 **else**
7 | One of the fail tests that executes the statement s is chosen to determine the correctness of the check point;
8 | Monitor the values of the variables before and after executing the check point variables CPV1 and CPV2;
9 | Remove statement(s) that are not relevant to the bug from the statement-based spectra coverage;
10 | Repeat Steps 1–11 until bug is found;
11 **end**

grammer, the algorithm stops. Otherwise, the programmer sets two other check points: before (CPV1) and after (CPV2) the check point of the statement s. A fail test that executes the check point of the statement s is randomly chosen. This test is used to determine all the related variables of the program that are related to the check point of the statement s. The correctness of the values of these variables are determined in CPV1 and CPV2 by the programmer. If the values of the variables are incorrect in CPV1, the bug is likely to be located before the check point CPV1 of the statement s. If the values of the variables are incorrect in CPV2, the bug is likely to be located after the check point CPV1 of the statement s. If both of the values of the variables in CPV1 and CPV2 are incorrect, the bug is likely located before the first check point of the statement s, CPV1. Based on these check points, statements which are not relevant to the bug are removed from the statement-based spectra coverage and will not be considered as part of the statement-based spectra coverage in the next iteration. The fuzzy set theory approach is then applied on the modified statement-based spectra coverage to generate a new ranking of statements [Hao et al., 2008]. Steps 1–11 are repeated until the programmer recognises that the statement s is the bug.

In their study, Hao et al. [2006] use the Siemens Test Suite to compare their proposed approach with the Dicing and Tarantula approaches [Agrawal et al., 1995, Jones and Harrold, 2005]. Hao et al. [2006] measure the effectiveness of their proposed approach by considering the number of check points needed to be examined by the programmer before the bug is found. As the check points are examined with respect to the program statements, this measure is comparable to the rank percentages measure (see Subsection

4.3.1). Their approach is able to locate 8.26% of the bugs in the Siemens Test Suite by examining less than 1% of the program code.

Ali et al. perform an investigation on the accuracy of bug localization performance [Ali et al., 2009] on Siemens Test Suite [SIR, 2010] and Space [Frankl and Iakounenko, 1998] where the bugs are hand-seeded by researchers and generated by an automated tool respectively. They claim that by using these fault-seeded datasets can be inaccurate in the study of bug localization approaches. They propose to evaluate bug localization performance on a benchmark known as Concordance which has naturally-occurring bugs (the details of this benchmark can be found in Section 4.5). They also use a mutant generator tool *mutgen* [Andrews et al., 2005] to generate mutants. A mutant results from making small changes in the source code of a program such as the removal of a statement, negating the logic of a conditional statement, and a logic-based error. These mutants are viewed as the bugs of the program. They evaluate the Concordance programs with both the naturally-occurring bugs and the seeded bugs (mutants) by the mutant generator.

Ali et al. [2009] use the performance measure of successful diagnosis of bugs, $SucDiag$ (see Subsection 4.3.2) to evaluate the effectiveness of bug localization performance. They observe that bug localization performance using the Tarantula metric on different types of bugs (naturally-occurring bugs and mutants) are quite similar. By examining less than 1% of the program code of the Concordance program, the programmer is able to locate 21% and 28% of the naturally-occurring bugs and the bugs generated using the mutant generator, respectively, in the Concordance program. They also apply a rule-based classification algorithm [Cohen, 1995] to locate bugs. They encounter the problem of class imbalance in their study, where most of the programs have more pass test cases than fail test cases. They propose a solution to this problem by applying a cost sensitive classification [Domingos, 1999] on the test cases using Weka [eibe, 2010]. Ali et al. [2009] observe an average of 1.40 for the accuracy of locating bugs in Concordance using the PART classification algorithm [Frank and Witten, 1998] with the cost sensitive classification.

Zoeteweij et al. propose an application of program spectra on real-world embedded software systems such as high-volume consumer electronic products [Zoeteweij et al., 2007]. They propose using the Jaccard metric [Chen et al., 2002] (refer to the metric in Table 2.3) to locate bugs in a program. They gather the block-based spectra coverage specifically on the load and lock-up problem of analog television set software. In the load problem, the CPU load is higher than usual after teletext viewing. In the lock-up problem, when the user searches in teletext pages (without visible content), the teletext system is locked up. The program code of 450K lines contains Koala software components [Van Ommering et al., 2000]. The load and lock-up problem only occur in a particular software version. They show that using the block-based spectra coverage approach

with the Jaccard metric helps to locate bugs in program code. However, there is a performance overhead on the block instrumentation as a result of using Front parser [Abreu et al., 2006].

Liblit et al. propose an approach that remotely sample predicates from user executions of program code [Liblit et al., 2003]. Typically, the user execution information is gathered in the form of bug reports before feeding them into a central database. This approach has been used in other software vendors such as Mozilla. In Mozilla, a typical bug report is gathered automatically from the user execution of the browser whenever the web browser crashes. The programmer can analyse and fix the bug using the information contained within these bug reports. In their study, Liblit et al. [2003] propose a sparse random sampling strategy to sample necessary predicates of the program code, instead of analysing all predicates of the program code from the user executions. The studies that consider the use of predicates to locate bugs can be found in Section 2.4. Pseudocode for their proposed approach is described in Algorithm 13.

Algorithm 13: Predicate Remote Sampling Algorithm [Liblit et al., 2003]

```
Input: predicate-based instrumented program code, sampled predicate countdown
       indicating next predicate to be sampled, current countdown
Output: sampled predicate(s)
1  Predicate Sampling;
2  foreach user execution of the program code do
3      if (next sampled predicate countdown > current countdown ) then
4          Fast Path;
5          Decrement the value of the sampled predicate countdown from the current
           countdown;
6          Go to the next sampled predicate countdown and perform sampling;
7      end
8      else
9          Slow Path;
10         Decrement the value of the sampled predicate countdown;
11         if next sampled predicate countdown reached zero then
12             Perform sampling on the predicate;
13             Reset and retrieve the next sampled predicate countdown;
14         end
15     end
16 end
```

Initially, the input to this algorithm is the predicate-based instrumented program code. Each predicate of the program code is assigned *sampled predicate countdown* and *current countdown* counters. The *sampled predicate countdown* indicates the next predicate to be sampled and the *current countdown* counter refers to the number of predicates to be sampled in the blocks of the program. If the next *sampled predicate countdown* counter is greater than the *current countdown* counter, this indicates that the current predicate should not be sampled. The current predicate is skipped, and the algorithm then hops

onto the *Fast Path* to reach to the next predicate to be sampled. The *sampled predicate countdown* counter is decremented. If the next *sampled predicate countdown* counter is not greater than the *current countdown* counter, the predicate will proceed to the *Slow Path*. As the next *sampled predicate countdown* reaches zero, sampling will be performed on the predicate. These steps are repeated for each user execution of the program code. Finally, the sampled predicates are returned to the programmer to analyse and locate the bugs. Liblit et al. [2003] observe that the sampled predicates generated using their proposed sparse random sampling algorithm are sufficient representation of the information to locate bugs. This approach does not affect the accuracy of locating bugs and has been used as the framework for the Liblit system [Liblit et al., 2005].

Liblit et al. [2003] propose several strategies to eliminate predicates deemed not to be buggy especially for the *return-value* predicate. This predicate has three possible return values; negative, zero, or positive values. A counter for each of the possible values for each predicate are tracked throughout the user executions. The predicate elimination strategies are detailed as the following:

1. Eliminate predicates whose counter is zero for any of the three return values in all user executions.

2. Eliminate predicates whose counter is zero for all of the three possible return values in all user executions that cause the program to fail.

3. Eliminate predicates whose counter is zero for any of the three return values in all user executions that cause the program to fail.

4. Eliminate predicates with a non-zero counter for the return values in the user executions which do not cause the program to fail.

Liblit et al. [2003] also propose to locate predicates using a machine learning approach: logistic regression (this is discussed further, along with other machine learning approaches, in Section 3.6). They use the CCRYPT [selinger, 2010] as their benchmark and found two predicates of the CCRYPT program are the potential bugs. They make an assumption that the bug is expected to be found in the predicates of the program. Bugs related to preprocessor directives or missing lines of code cannot be located using their approach.

Chung et al. [2008] propose a similar fuzzy set theory approach as Hao et al. [2008], using predicate-based spectra coverage instead of statement-based spectra coverage. Pseudocode of their proposed approach is presented in Algorithm 14. The input to this algorithm is the predicate-based spectra coverage. The coverage is the test coverage information (in binary form) of the predicates of a program which are *True* (represented in the

matrix E) and the predicates of a program which are executed (represented in the matrix E′). The details of predicates where they are *True* and being executed have been described by Liblit et al. [2005]. We have shown an example of these predicates in Figure 2.1. Initially, Chung et al. [2008] apply the fuzzy set theory approach [Hao et al., 2008] on the matrices E and E′. The computed membership grade matrices from the matrices E and E′ are represented by the new matrices M and M′ respectively. This is followed by finding the influence of each predicate. The corresponding m_{st} element (computed membership grade of the element) in the matrix M is subtracted from the m'_{st} element in the matrix M′ using Definition 6. Predicates such as *if*, *while*, and *for* have more influence due to the differences in membership grades contained within M and M′. The computed influence matrix is denoted by matrix T. Several heuristics are defined to refine the selection of predicates in the matrix T. The fine details of these heuristics can be found in their study [Chung et al., 2008]. This refinement step ensures that predicates are ranked accurately. The refined matrix is mapped to the matrix D, and the predicates are ranked using the similar suspicious probability function, $P(s)$ [Hao et al., 2008].

Algorithm 14: Algorithm of Bug Localization of Predicates using Fuzzy Set Theory Approach [Chung et al., 2008]

Input: Matrix E of predicate-based spectra coverage where predicate is *True*, matrix E′ of predicate-based spectra coverage where predicate is executed
Output: ranked predicates of the program based on suspicious probability function, $P(s)$

1 **Compute membership grade matrix**;
2 Fuzzy set theory [Hao et al., 2008] is applied on matrix E and E' respectively to obtain matrix M and M' respectively;
3 **Calculate the influence of each predicate using the matrix M and M' (T←M−M')**;
`// mst and m′st refers to the statement s of test case t in matrix M and M′ respectively`
4 **foreach** $m_{st} \in M \wedge m'_{st} \in M'$ **do**
5 Apply Definition 6;
6 **end**
7 **Refine the existing influence of each predicate**;
8 D←T using several heuristics;
9 **Compute importance value of each predicate**;
10 Suspicious probability function, $P(s)$ [Hao et al., 2008] is performed on the new matrix, D;
11 Rank predicates according to the suspicious probability function;

Definition 6 (Influence)**.**

$$T = \begin{cases} |m_{st} - m'_{st}| & \text{if } m_{st} \neq m'_{st} \\ -1 & (\mathrm{m_{st}} = \mathrm{m'_{st}}) \wedge (\mathrm{m_{st}} > 0) \\ 0 & \text{otherwise} \end{cases}$$

Chung et al. [2008] evaluate their proposed approach on five small-size C programs, namely: *f_sum.c*; *nre_fib.c*; *checkID.c*; *big_sum.c*; and *sdes.c*. They measure the effec-

tiveness of their proposed approach on the basis of the number of predicates examined before a bug is found divided by the total number of predicates in the program. In order to make a fair comparison with other approaches, such as the fuzzy set theory approach on the statements of the program [Hao et al., 2008] and the Dicing approach [Agrawal et al., 1995], Chung et al. [2008] propose to measure the effectiveness of their approach by computing the number of program statements needed to be examined before the bug is found. By using their approach, the programmer only needs to examine in the range from 7.35% to 42.11% of the program code to locate the bugs as compared to the range from 10.29% to 57.89% of the program code using the fuzzy set theory approach on the statements of the program [Hao et al., 2008].

Zhang et al. propose to use short circuit evaluation within the predicates of a program to locate bugs [Zhang et al., 2008]. They represent these short circuit evaluation sequences using Boolean expressions. An example of a short circuit evaluation is $*$j $<= 1$ || src [$*$i+1] == '\0'. This approach is known as **D**ebugging **E**valuation **S**equences (DES). Different evaluation sequences can be formed from the short circuit evaluation and are represented as Boolean expressions. The pseudocode of their proposed approach is presented in Algorithm 15. The input to this algorithm is the predicate-based instrumented program code. Initially, all possible evaluation sequences of the program are identified. Each evaluation sequence is treated as a distinct predicate. Zhang et al. [2008] apply the CBI and SOBER systems to assign values to these predicates. The studies of the CBI [Liblit et al., 2005] and SOBER [Liu et al., 2005] systems have been described in Chapter 2. Finally, predicates are ranked on the basis of the values assigned to them. The application of their proposed evaluation sequence method (DES) on the CBI and SOBER approaches forms DES-CBI and DES-SOBER approaches respectively.

Algorithm 15: Algorithm of Debugging Evaluation Sequences (DES) Approach [Zhang et al., 2008]

```
   Input: predicate-based instrumented program code
   Output: ranking of respective predicates
1  foreach instrumented predicate do
2      Identify all possible short circuit evaluation sequences for respective predicates;
3      for each evaluation sequence do
4          Treat every evaluation sequence as a distinct predicate;
5          Use CBI [Liblit et al., 2005] and SOBER [Liu et al., 2005] to assign value to the
           predicate;
6      end
7  end
8  Rank predicates in descending order according to predicates likely to be buggy;
```

Zhang et al. [2008] also evaluate the CBI [Liblit et al., 2005] and SOBER [Liu et al., 2005] approaches. In order to compare their proposed approach with the latter predicate-based approaches, they only consider the first five distinct predicates of the evaluation

sequences that are ranked top as the predicates likely to be buggy. Zhang et al. [2008] also consider the first five predicates that are ranked highest of the CBI [Liblit et al., 2005] and SOBER [Liu et al., 2005] approaches. They evaluate their proposed approach on the Siemens Test Suite and observe that their approach is able to locate more bugs when compared to the CBI and SOBER approaches. In their study, Zhang et al. [2008] determine the relative effectiveness of their approach with the former studies with respect to the percentages of bugs located in the test suite. They compare the percentages of bugs that are able to be located using SOBER or CBI when compared to DES-SOBER or DES-CBI approaches, respectively. By examining 10% of the program code, 20% of the bugs are able to be located as compared to 10% of the bugs in the Siemens Test Suite when using the DES-SOBER and SOBER approaches respectively. By examining the same percentage of program code, 24% of the bugs are able to be located as compared to 14% of the bugs in the Siemens Test Suite when using the DES-CBI and CBI approaches respectively. The latter findings indicate that evaluation sequences, which are treated as predicates of a program, is a useful approach to locate bugs effectively.

3.4 State-based Approaches

In this section, we describe several state-based approaches in which the states of the program being debugged are used to locate its bugs. In the traditional debugging approach, the programmer often needs to use a specific debugger, with respect to the compiler, in order to locate the bugs in a program. The programmer has to traverse each statement of the program code (buggy region) to observe the states of the program in order to understand and locate its bugs. There are several studies that propose to automate the steps involved in the traditional debugging approach in order to locate the bugs in a program.

Zeller has proposed an automation of the debugging approach, the resulting technique is known as automated debugging or Delta Debugging [Zeller, 2000]. The goal of the automated debugging approach is to isolate failure inducing circumstances. Failure inducing circumstances refer to the test case inputs that cause a program to fail (crash). These circumstances are as follows:

1. Program input e.g input of webpage causing the web browser to fail (crash).

2. User interaction e.g keystrokes of the user causing the program to fail (crash).

3. Changes to the program code e.g failure inducing code changes in regression testing.

Algorithm 16 presents the pseudocode for isolating the failure inducing circumstance of the *program input*. In this failure inducing circumstance, the test case input refers to the HTML code that causes a web browser to fail (crash). For example, the HTML code of <SELECT NAME="priority" MULTIPLE SIZE=7> that causes a web browser to fail (crash). The test case input (HTML code) needs to be simplified one at a time and rerun with the web browser to check whether the web browser fails (crashes). The algorithm starts to remove a large chunk of the test case input (HTML input) before narrowing down to remove a few more characters of the test case input. If the web browser does not fail (crash), the algorithm stops immediately and returns to the programmer the test case input that previously caused the web browser to fail (crash). If the web browser still fails (crashes), the test case input is further simplified and the test is rerun to check whether the web browser fails (crashes). This step is repeated until the simplified test case input does not cause the web browser to fail (crash). Zeller [2000] evaluates Mozilla as the benchmark in his study and it takes about 21 minutes on average to isolate a failure inducing circumstance of *program input* in a web page.

Algorithm 16: Algorithm of Isolating Program Input that Causes Failure to the Web Browser [Zeller, 2000]

Input: a test case input that causes failure to the web browser
Output: test case input that last causes failure to the web browser
1 **while** *test case input can be further simplified* **do**
2 | Load web browser using the simplified test case input;
3 | **if** *the simplified test case input causes failure (crash) to the web browser* **then**
4 | | Simplify the test case input;
5 | **end**
6 | **else**
7 | | Stop and go to Step 11;
8 | **end**
9 **end**
10 Test case input could not be further simplified (input is minimum);
11 Return the last test case input as the root cause of the failure (crash) of the web browser;
12 Exit;

The approach of simplifying test case input is used to debug the failure inducing circumstance of *user interaction*. In the failure inducing circumstance of *changes to the program code*, Zeller [2000] proposes using test case input that does not cause a program to fail (pass test input) and test case input that causes the program to fail (fail test input). These test case inputs are used to isolate the bug of the program simultaneously. He observe that his algorithm requires extensive computation in order to isolate the failure inducing circumstances.

Zeller extends his study by proposing another automated debugging approach, better known as cause-effect chain [Zeller, 2002]. Instead of narrowing down the test case input that causes program failure, he proposes the use of memory graphs [Zimmermann and

Zeller, 2002] to narrow down the states (variables and values) of the program being debugged that cause program failure. In this approach, he uses two different versions of the program code (one that does not cause program failure, P, and one that causes program failure, P′). Memory graphs represent the states of a program in graphical form – consist of all the values and variables of the program (e.g pointer referencing). A memory graph extractor is used to generate the memory graphs of a program, the details of which can be found in Zimmermann et al. [2002].

Algorithm 17: Algorithm of Cause-Effect Chain [Zeller, 2002]

Input: program code version that does not cause program failure P, program code version that causes program failure P′
Output: cause-effect chain of failure-induced states that are relevant to the failure

1 **Isolate relevant failure inducing state**;
2 Use GDB [GCC, 2010b] to extract all the states of the program code P and P′;
3 **foreach** *state* **do**
4 Extract the memory graphs of the states of both P and P′;
5 Make comparison of the vertices and edges from the memory graphs;
6 Variables of the state that causes failure are treated as inputs for Step 7;
7 Apply these inputs to P′ using Delta Debugging approach (see Algorithm 16);
8 Obtain the last input (variables of the state) that causes failure to P′;
9 **end**
10 Gather all the last input (variables of the states) as part of the cause-effect chain;

The pseudocode of the proposed approach by Zeller [2002] is described in Algorithm 17. The input to this algorithm is the two different versions, P and P′ of a program, as described previously. Initially, GNU GDB debugger [GCC, 2010b] is used to extract the relevant states of both versions of the program code. In each of the states, the memory graphs of both versions of the program code are extracted using the memory graph extractor of Zimmermann et al. [2002]. The vertices and edges within these graphs are compared. The differences in these vertices and edges, consisting of variables and values, form the inputs of the Delta Debugging approach [Zeller, 2000] (see Algorithm 16). This approach is applied to isolate the failure-induced input (variables of the states) that are relevant in causing the program to crash. The variables of the states that last caused failure (crash) to the program code are obtained. These variables are gathered for each state of the program, and are returned as the cause-effect chain of the failure of the program. The states (variables and values) in the cause-effect chain are presented to the programmer as the root cause of the program failure.

Zeller [2002] uses programs in GNU GCC [GCC, 2010a] to evaluate and demonstrate the effectiveness of his proposed approach. His proposed cause-effect chain approach does not need complete information of the program code. The entire framework has been made public in the AskIgor server [Zeller, 2010]. However, this server has been closed recently to the public due to the high maintenance costs of the framework.

Zeller [2002] proposes using the cause-effect chain approach in order to discover the state of a program that causes its failure [Zeller, 2002]. The resulting approach is known as *searching in space*. Cleve and Zeller propose a state-based approach by identifying the transition time between the point at which a variable of the program stops causing failure and another variable of the program is causing failure [Cleve and Zeller, 2005]. They are interested to find the causes of a failure in the cause-effect chain by *searching in time* instead of *searching in space*. The *time* or *moment* where the variables of the program change and cause program to fail indicates the variables are the bug.

Algorithm 18 describes the algorithm, designed by Cleve et al. [2005] to isolate cause transitions of the states that cause the program to fail. The algorithm consists of three major phases: the first is isolating failure inducing states, followed by finding cause transitions in the states, and finally *searching in time* to isolate the cause transitions of each state. Initially, they use GDB [GCC, 2010b] to extract all the states of the program code on both pass and fail test cases. Then, they apply Delta Debugging [Zeller, 2000] in order to isolate the failure-inducing (state) differences between the test cases that are relevant to the program failure. The output of this phase is the *cause and effect chain* (see Algorithm 17) [Zeller, 2002].

This cause and effect chain contains the first and the last state that cause the program to fail. From these chains of states, the search for any direct cause transitions between these two relevant states can be narrowed down. The direct cause transition can be a function, a variable, or a program statement where the *moment* or the *time* where the bug is detected. Cleve et al. [2005] propose to monitor the time where variables of a particular state change and cause program failure (detailed in Algorithm 19). Finally, the divide and conquer algorithm is applied to locate the exact time when the initial and the corresponding state of the program (variable) changes and causes program failure. For each cause transition, the first and final state of the program are examined to identify more cause-transitions between these states. This phase is performed iteratively until all the cause-transitions among the subsets of the states have been examined.

In their study, Cleve et al. [2005] evaluate GNU GCC and the Siemens Test Suite in order to measure and compare the effectiveness of their proposed approach with Renieres et al. [2003] by using the program dependence graphs (PDG). Cleve et al. [2005] use the same measure as Renieres et al. [2003] (see Definition 12 in page 78). It refers to the number of nodes (statements) not needed to be examined before the buggy node is found. By examining less than 1% of the program nodes, their proposed approach is able to locate 4.65% of the bugs as compared to none of the bugs in the Siemens Test Suite when using the approach of Renieres et al. [2003]. The proposed approach by Cleve et al. [2005] is able to locate more bugs than the latter for all the ranges of node percentages –

Algorithm 18: Algorithm of Locating Causes of Program Failure [Cleve and Zeller, 2005]

Input: pass test case, fail test case, memory graph of the pass test case, memory graph of the fail test case
Output: Subset of program states that have cause transitions

1 **Isolating Failure-Inducing States**;
2 Use GDB [GCC, 2010b] to extract all the states of the program code on both pass and fail test cases;
3 Isolate failure-induced states on both of the test cases using Delta Debugging approach [Zeller, 2000];
4 **Finding Cause Transition in States**;
5 **foreach** *state of the program* **do**
6 Apply Cause-Transition in Algorithm 19 to find the direct cause transition from one state to the other state of the program;
7 **end**
8 **Isolating Cause Transitions**;
9 **foreach** *cause transition found in Steps 4–7* **do**
10 **Using divide and conquer algorithm**;
11 Obtain the first state and final state (fail state) where cause transition is found;
12 Examine the state at middle of the interval between the two states;
13 Examine whether cause transition has occurred already in first or second part of the states;
14 **if** *Cause Transition found in either first part or second part of the states* **then**
15 Gather the intermediate state as the new state;
16 Repeat Steps 4–7 to find more cause-transitions in the subset of states;
17 **end**
18 **end**

Algorithm 19: Cause-Transition Algorithm

Input: variables and values of subset of states found in Algorithm 18
Output: the variables and values from the subset of states that cause program failure

1 **foreach** *moment of time,*t_f *and* t_l **do**
// t_f as the time where first state is found, t_l as the time where final state is found
2 Apply the first and last variables on the current state of the program code;
3 Monitor the outcome of the variables during the time period;
4 **if** *outcome of the variables cause program failure* **then**
5 Cause-Transition occur;
6 **end**
7 **else**
8 No Cause-Transition found between the variables of these states;
9 **end**
10 **end**

the percentages of nodes that need to be examined by the programmer to locate the buggy node (statement).

3.5 Test Reduction Approaches

We describe, in this section, several studies that use test reduction approaches with respect to bug localization performance. In this thesis, we also study the use of test suite reduction (using unique test cases) which can be found in Chapter 7.

Wong et al. perform a study on the effectiveness of bug localization performance by removing redundant test cases [Wong et al., 1994]. An implicit enumeration algorithm [Etcheberry, 1977] is used in their study to find the optimal set covering of non-redundant test cases (reduced test suite) without compromising the test coverage of the entire test suite (unreduced test suite). Wong et al. [1994] use the ATACMIN tool [Horgan and London, 1992] to obtain the reduced test suite.

Wong et al. [1994] evaluate their proposed approach on the Unix Test Suite, which consists of 10 programs. Initially, they generate test suites of the programs with different block coverage ranging from 50%-55%, 60%-65%, 70%-75%, and 80%-85%. They apply their proposed reduced test suite approach on these test suites of the 10 programs. They evaluate and compare bug localization performance of the programs using the unreduced test suite and reduced test suite. A slight improvement in the bug localization performance is observed when using the reduced test suite. They refer to the bug localization performance as the ratio of the number of programs that successfully detect the bugs divided by the total number of programs evaluated. A program is considered to have successfully detected the bugs if one of the test cases in the program executes the bugs. They observe no clear relationship between the size of the reduced test suite and bug localization performance on the reduced test suite. They extend this study on different datasets and case studies [Wong et al., 1998].

Yu et al. perform an empirical study of the test suite reduction with respect to bug localization performance [Yu et al., 2008]. They evaluate two types of reduction approaches, namely: the statement-based and vector-based reduction. The statement-based reduction classifies a test case as redundant if its test coverage is the subset of another test case. Vector-based reduction classifies a test case as redundant in the event that another test case has identical coverage. We show an example of both statement-based and vector-based reduction approaches in Table 7.1. Yu et al. [2008] introduce different reduction approaches for pass and fail test cases on the Siemens Test Suite and Space programs [Do et al., 2005]. The details of the different reduction approaches can be found in their study [Yu et al., 2008].

Algorithm 20: Algorithm to Generate Subset of the Unreduced Test Suites

Input: statement-based spectra coverage labelled with Pass and Fail
Output: statement-based spectra coverage that forms the subset of unreduced test suite of size M

1 Randomly select one fail test case from statement-based spectra coverage labelled with Fail;
2 **while** $M \leq 500$ **do**
3 $M \leftarrow M$+50 // M as number of test cases set to 0, N as counter variable set to 1
4 **foreach** *test case,* t *to be selected and counter* $N < M$ **do**
5 Randomly select one test case from existing test cases regardless of Pass/Fail;
6 Mark the selected test case so as not to select again in next iteration;
7 $N \leftarrow N + 1$;
8 **end**
9 Return the unreduced test suite of respective size M;
10 **end**
11 Repeat Steps 2–10 for 100 times to generate 100 different test suites with respect to unreduced test suite of size M;

In Chapter 7, we propose to evaluate unique test cases with respect to bug localization performance. We use the entire test suite (unreduced test suite) before removing the redundant test cases in order to obtain the unique test cases. Yu et al. [2008] also performed the study of using reduced test suite with respect to bug localization performance. However, they use the subset of the unreduced test suite before removing the redundant test cases to obtain the unique test cases. They generate subset of the unreduced test suites of different sizes ranging from 50 test cases to 500 test cases. The algorithm used to generate the subset of the unreduced test suite is presented in Algorithm 20. The input to this algorithm is the statement-based spectra coverage for the entire test suite of the program. A fail test case is randomly selected from the entire test suite of the program to ensure the bug can be located by at least one fail test case. This is followed by selecting test cases randomly from the entire test suite of the program regardless of the test being a pass or a fail test. This selection of test cases is performed for different sizes of unreduced test suite, size M, ranging from 50 test cases up to 500 test cases. The selected test case subsets form the unreduced test suite of size M. Steps 2–10 are repeated 100 times for the unreduced test suite of size M to reduce any bias incurred during test case selection. Yu et al. [2008] evaluate the two types of reduction approaches, namely the statement-based reduction and vector-based reduction methods.

In their study, Yu et al. [2008] use the Tarantula [Jones and Harrold, 2005], CBI [Liblit, 2004], Jaccard, and Ochiai metrics [Abreu et al., 2006] to evaluate the effectiveness of their proposed approach in terms of bug localization performance. They use the rank percentages measure (Definition 11) to evaluate their approach using both unreduced and reduced test suites of the 169 programs in the Siemens Test Suite and Space. They com-

pare the percentages of the program code needed to be examined by the programmer before the bug of the program is found.

Definition 7.

$$Reduction = (1 - \frac{\textit{number of test cases in reduced test suite}}{\textit{number of test cases in unreduced test suite}}) * 100\%$$

Yu et al. [2008] also introduce the $Reduction$ metric (see Definition 7) to observe the ratio of the number of the test cases in the reduced and unreduced test suites. By using this metric, they observe that more test cases are reduced using the vector-based reduction approach than the statement-based reduction approach. Larger $Reduction$ indicates that more redundant test cases are in the test suites.

Some programs of the Siemens Test Suite and Space show improved bug localization performance on all the spectra metrics by using the vector-based reduction approach. The statement-based reduction approach shows more variation in bug localization performance across the different programs of the Siemens Test Suite and Space when compared to the vector-based reduction approach. Yu et al. [2008] conclude that by using the Tarantula metric, the bug localization performance of the vector-based reduction approach outperforms that of the statement-based reduction approach. The programmer needs to examine more program code in order to locate bugs in the program using the statement-based reduction approach when compared to using the vector-based reduction approach. The improvement in bug localization performance using the vector-based reduction approach relative to the unreduced test suite for 169 programs, for different test suite sizes and reduction variants, ranges from 0.06% to 0.11%. By using the statement-based reduction approach on these programs, Yu et al. [2008] observe drop in bug localization performance compared to using the unreduced test suite, in the range of 0.58% to 8.32%.

Hao et al. conduct an experimental study of vector-based reduction approach with respect to the bug localization performance [Hao et al., 2005]. They apply the Dicing [Agrawal et al., 1995] and the Tarantula [Jones et al., 2002] approaches to evaluate the bug localization performance obtained when using the unreduced and reduced test suites. In their study, Hao et al. [1995] use the Desk Calculator (DC) [Morris and Cherry, 1983] and Tiny C Compiler (TCC) [Bellard, 2010] programs. They show that bug localization performance improves when using the Dicing and Tarantula approaches on the reduced test suites (non-redundant test cases) for the DC program. The average improvement in bug localization performance on the reduced test suite, when compared to using the unreduced test suite of the DC is 1.22% and 2.08% for the Dicing and Tarantula approaches respectively. However, the Tarantula metric does not result in an improvement in bug localization performance when using the reduced test suite relative to using the unreduced

test suite on the TCC program. There is a slight drop of 0.23% on average, in bug localization performance using the Tarantula metric on the reduced test suite of the TCC program when compared to using the unreduced test suite.

3.6 Combining Spectra-based and Machine Learning Approaches

Recently, several data mining and machine learning approaches have been proposed with the spectra-based approach to locate bugs in the program code effectively. In this section, we describe several studies that use these approaches.

Jones et al. propose a parallel debugging technique to locate bugs in multiple-bug programs [Jones et al., 2007]. Fail test cases that are responsible for respective bugs can be distributed to multiple programmers – each programmer debug the program simultaneously. This approach can improve the time taken to locate the bugs in a program. However, most of the time, a bug is caused by more than one fail test case. Therefore, Jones et al. propose to cluster fail test cases that are responsible for a particular bug. They propose two techniques to generate clusters for the fail test cases:

1. Clustering based on the profiles and bug localization results, $Cluster_1$.
2. Clustering based on the bug localization results, $Cluster_2$.

Initially, the program code is instrumented with program statements. Pseudocode for the first technique is presented in Algorithm 21. The input to this algorithm is the statement-based spectra coverage of the fail test cases, represented in a binary form (similar to Table 2.2). The sequences of the branches of the program (consisting of statements) executed in the fail test cases are represented in discrete-time Markov chains (DTMC) [Romanovskiĭ, 1970] – better known as behaviour models. This is followed by applying the hierarchical clustering algorithm [Murtagh, 1983] on the DTMC of the fail test cases. The output of the hierarchical clustering algorithm is usually represented in a graphical diagram known as dendogram [van Rijsbergen, 1979]. For each level of the dendogram, a new cluster is formed by comparing the similarity of the DTMC (sequences of the branch profiles) of the fail test cases. The similarity of any two DTMCs are considered in terms of the sum of the absolute difference of the branch profiles in these DTMCs. This step is repeated until only one cluster is formed in the dendogram.

Based on the clusters formed, Jones et al. [2007] use the bug localization results to stop clustering and refine the clusters of fail test cases. The clusters on the different levels of the dendogram may belong to each other and can be merged. For each level of the dendogram,

Algorithm 21: Algorithm of Clustering based on Profiles and Bug Localization Results, $Cluster_1$ [Jones et al., 2007]

Input: a set of instrumented statement-based spectra coverage of the fail test cases
Output: cluster(s) of fail test cases responsible for the respective bug(s)
1 Identify and gather sequences of branch profiles (consist of statements) executed in fail test cases and represent in discrete-time Markov chains (DTMCs);
2 Apply hierarchical clustering algorithm on the DTMC for respective fail test cases;
3 **foreach** *level of the dendogram* **do**
4 Choose the smallest absolute differences of the branch profiles in each pair of DTMC;
5 Treat these fail test cases as one cluster;
6 **if** *only one cluster formed for the level* **then**
7 Stop;
8 **end**
9 **end**

specialised test suites are formed for the clusters of fail test cases. A typical *specialised test suite* consists of the fail test cases and the set of all pass test cases. The test suite contains the statement-based spectra coverage of the fail test cases and the set of all pass test cases. They evaluate the *specialised test suite* using the Tarantula metric and produce the bug localization result (similar to Jones et al. [2005]). The ranking of the program statements for the *specialised test suite* is then generated. Jaccard metric is then used to compare the similarity of the bug localization result (the ranking of program statements) of pairs of the *specialised test suite* in the dendogram. They also set a threshold in the comparison of the *specialised test suite*. If the similarity of the pairs of any two clusters of the *specialised test suite* is within the threshold, these clusters are merged.

Algorithm 22: Algorithm of Clustering based on Bug Localization Results, $Cluster_2$ [Jones et al., 2007]

Input: fail test cases, *specialised test suite*
Output: cluster(s) of fail test cases responsible for respective bug(s)
1 Use Tarantula metric to evaluate and rank all the statements for each *specialised test suite*;
2 Use Jaccard metric to perform pairwise similarity on the ranking statements between the specialised test suites;

The second clustering technique (see Algorithm 22) is a much simpler and straightforward approach where fail test cases are clustered based on bug localization results. The inputs to this algorithm are the set of fail test cases and the *specialised test suite*. Each *specialised test suite* contains the statement-based spectra coverage of a fail test case and all the pass test cases. The *specialised test suite* is evaluated using the Tarantula metric and the statements of the program that are likely to be buggy are ranked. The Jaccard metric is used to compare the ranked statements between all the pairs of the *specialised test suites*. Jones et al. [2007] use a threshold in order to cluster the fail test cases that are similar and responsible for a particular bug of the program.

Jones et al. [2007] evaluate the effectiveness of their proposed bug localization ap-

proaches using the rank percentages measure (Definition 11). They compare the cost of debugging using the sequential and parallel modes. The sequential mode requires the programmer to examine the program code from the top of the ranked list, assuming there is only one bug is being searched for at a time [Jones and Harrold, 2005, Abreu et al., 2006, Wong et al., 2007]. In the parallel mode, the two clustering approaches defined in the Algorithm 21 and Algorithm 22 are used. They conduct their evaluation using the Space program [Do et al., 2005] and create 100 8-bug versions of the Space programs. They observe that using the sequential mode takes more time, relative to the parallel mode, to locate bugs. They consider the relative effectiveness of locating bugs for multiple-bug programs using the proposed clustering approaches in parallel mode and the sequential mode. In their study, the average rank percentages of locating bugs in the sequential mode, parallel mode ($Cluster_1$), and parallel mode ($Cluster_2$) for 90 multiple-bug Space programs using the Tarantula metric is 36.63%, 31.50%, and 26.43% respectively.

Denmat et al. [2005] reinterpret the Tarantula metric [Jones and Harrold, 2005] using the association rule measures in data mining such as Confidence and Support [Brin et al., 1997]. Denmat et al. [2005] relate the term *transaction* used in the data mining community to test cases. They define X as the buggy statement that causes the program to fail and Y as the fail test case. They define *Confidence* as the probability of having the program fail given that the buggy statement has been executed by the fail test cases. They also define *Support* as the probability of the test cases that cause program to fail. Denmat et al. [2005] define *Lift* metric (Definition 8) as the ratio of *Confidence* and *Support*. They formally prove that *Lift* is equivalent to the Tarantula metric (see the metric in Table 2.3).

Definition 8.

$$Lift(X \rightarrow Y) = \frac{Confidence(X \rightarrow Y)}{Support(Y)}$$

Denmat et al. [2005] discuss the following limitations of using the Tarantula metric to locate a buggy statement:

1. The Tarantula metric is unable to locate a bug effectively if the buggy statement is a missing statement or a missing case in a switch statement.

2. The bug can be located more precisely if the program is instrumented with program blocks instead of program statements.

3. Buggy statements in the program code cannot be located using the Tarantula metric if they belong to the statement initialisation (always executed by all the test cases).

4. Program code with multiple bugs cannot be located effectively using the Tarantula metric.

Briand et al. propose a machine learning technique, the C4.5 algorithm, to generate decision trees as a means of identifying multiple-bugs of program code [Briand et al., 2007]. They propose **RU**le-**BA**sed statement **R**anking algorithm (better known as RUBAR). Pseudocode for this algorithm is presented in Algorithm 23. The inputs to this algorithm are test specifications, pass and fail test cases, and the program code.

Algorithm 23: Algorithm of RUBAR using Category-Partition Approach [Briand et al., 2007]

```
   Input: test specifications, pass and fail test cases, program code
   Output: program statements with respective weights assigned
 1 Use Category-Partition method to generate transformed test cases from the test
   specifications;
 2 Classification;
 3 Generate rules from the transformed test cases using the C4.5 algorithm;
 4 Classify rules to Pass and Fail rules;
 5 Compute weights for Pass and Fail rules;
 6 foreach rule do
 7     if Pass rule then
 8         Statements related to the rule are assigned positive weight;
 9     end
10     else
11         Statements related to the rule are assigned negative weight;
12     end
13 end
14 Weights aggregated for statement and presented to the programmer;
```

Briand et al. [2007] claim that by using test cases, the bugs of a program code cannot be identified accurately. They propose the use of a category partition method in order to transform test cases using the test specifications of the program [Ostrand and Balcer, 1988]. The details of this transformation of test cases can be found in the Ostrand and Balcer [1988]. Using this method, test specifications are decomposed and transformed into test cases. These test cases contain more information about the conditions that could potentially be the program bug. An example of conditions in the transformed test case is the size of an array and whether this array is empty. The C4.5 classification algorithm [Quinlan, 1993] is applied on the transformed test cases to generate rules, which are represented as decision trees. Each rule consists of a set of statements of the program and is classified as Pass or Fail. In each rule, the classification is performed using pass and fail test cases that execute the set of the statements of the rule. A rule is denoted as Fail rule if the majority of Fail tests execute the set of statements of the rule. A rule is denoted as Pass rule if the majority of Pass tests execute the set of statements of the rule. Statements related to the Pass and Fail rules are assigned positive and negative weights respectively. The weight of each statement s is then aggregated across all the rules, R and is computed using Definition 9. In this definition, for each statement s, the ratio of the number of pass and fail test cases executing the statement s is aggregated across all the rules. Statement

s with the lowest weight value is deemed likely to be buggy as it is usually only found in the Fail rules but not in the Pass rules.

Definition 9.

$$Weight_s = \sum_{rule \in R} \frac{number\ of\ pass\ tests\ executing\ s - number\ of\ fail\ tests\ executing\ s}{number\ of\ test\ cases\ executing\ s}$$

Briand et al. [2007] evaluate the Space program [Do et al., 2005] and combine the bugs in all Space program versions into a single program with these bugs. There are 34 buggy statements in the Space program. Briand et al. [2005] use the precision and recall measures of Olson et al. [2008] to determine the effectiveness of their proposed approach. Using the C4.5 decision tree algorithm (RUBAR approach), the fail test cases are able to yield precision and recall of 91.7% and 94.7% respectively. They compare their proposed C4.5 algorithm (RUBAR) and the Tarantula approaches [Jones and Harrold, 2005]. By only examining 10% of the program statements in the Space program, 15% and 25% of the bugs can be located in the Space using the Tarantula and their proposed RUBAR approach respectively. This shows that their proposed C4.5 algorithm (RUBAR approach) is able to locate multiple-bug programs effectively compared to using Tarantula approach.

Algorithm 24: Algorithm of Finding Failures by Cluster Analysis Approach [Dickinson et al., 2001]

Input: Instrumented function caller/callee of the fault-seeded program code version, test cases
Output: clustered test cases
1 Execute test cases on the fault-seeded program code to gather function caller and callee of the execution coverage;
2 **Using hierarchical clustering algorithm [Murtagh, 1983]**;
3 **foreach** *iteration or level of dendogram* **do**
4 Apply similarity measures on the function caller and callee execution coverage of test cases;
5 Execution coverage of test cases that are similar and fulfill the minimal dissimilarity threshold forms a cluster;
6 **end**

Dickinson et al. propose to cluster test cases based on the similarity of test execution coverage [Dickinson et al., 2001]. In this study, the percentages of failure in the clusters is evaluated. The percentages of failure in the clusters refers to the percentage of fail test cases that are found in each cluster. If fail test cases are in each of the clusters, an assumption is made that the bugs of the program can be located in these clusters. Dickinson et al. instrument the fault-seeded version of the program code with respect to function-based spectra coverage. A brief pseudocode of their proposed approach is described in Algorithm 24. The inputs to this algorithm are the function-based instrumented program code and the test cases. Initially, test cases are executed with the program code to gather

the execution coverage of functions such as function caller and callee. This execution coverage is generated using the Sun^{TM} Java virtual machine.

This is followed by clustering similar test cases using the hierarchical clustering algorithm [Murtagh, 1983]. For each level of the dendogram generated using the hierarchical clustering algorithm, there are several different types of similarity measures which can be applied on the function caller and callee of the test execution coverage to form test case clusters. An example of a similarity measure is Euclidean distance. A minimal dissimilarity threshold is set in order to limit the number of iterations to obtain the relevant number of clusters. Test cases which exhibit similar Euclidean distance and fulfill the minimal dissimilarity threshold, form a cluster. The details of other similarity measures can be found in Dickinson et al. [2001].

After the clusters are formed, sampling is performed on the clusters to observe the effectiveness of finding failures within them. There are three types of sampling strategies, namely the *one-per-cluster sampling*, *adaptive sampling*, and *random sampling*. By using the *one-per-cluster sampling* approach, a test case is selected for each cluster. The total number of test cases selected using this approach is equivalent to the number of clusters. The *adaptive sampling* approach is similar to the *one-per-cluster sampling* with the exception that the selected test case of the cluster is checked to determine if it causes the program to fail. If the selected test case causes the program to fail, all of the other test cases in the cluster are selected regardless of being classified as pass or fail. The *random sampling* approach is used to compare the two sampling approaches just described. In the *random sampling* approach, a specified number of test cases are chosen randomly. Dickinson et al. [2001] report the effectiveness of finding failures using all the three sampling approaches with respect to the size of the clusters. For each cluster size, they report the average percentages of the fail test cases in the cluster divided by the total number of fail test cases of the program.

Dickinson et al. [2001] evaluate their approaches on several Java programs and GNU GCC version 2.95.2. In their evaluation, they observe that bugs are not distributed in a random fashion. More than 50% of the bugs can be found in the smallest cluster. On average, in the smallest cluster for the Java and GCC programs, 40.26% and 5.55% of the failures can be located as compared to 4.30% and 1.12% of the failures using the *adaptive sampling* and the *one-per-cluster sampling* approaches, respectively.

Di Fatta et al. propose using a frequent pattern mining algorithm in order to locate buggy functions in a program [Di Fatta et al., 2006]. Pseudocode for this approach is presented in Algorithm 25. The input to this algorithm is the function-based spectra coverage. The pass and fail test cases in the function-based spectra coverage are then represented

in function call tree. A function call tree refers to the set of functions (represented by vertices and edges) that are executed in the test cases.

The algorithm is divided into three major phases. Initially, abstraction is performed on the function call trees. Function call trees can cause performance and memory overhead due to many loops and iterations being present in functions of the program. Therefore, *zero-one-many* abstraction has been proposed in order to reduce the number of function call trees of the program. Take, for example, if there is a function tree of the program loop that has been executed more than once, the function call tree of the loop are abstracted twice. This can substantially reduce the memory overhead of storing the function call trees of the program.

Algorithm 25: Algorithm of Discriminative Patterns using Frequent Pattern Mining Approach [Di Fatta et al., 2006]

Input: function-based spectra coverage represented in function call trees
Output: functions ranked according to how likely they are buggy

1 **Abstraction phase**;
2 Function call trees are analysed using *zero-one-many* abstraction;
3 **Filtering phase**;
4 Define neighbourhood size N;
5 Use Frequent Pattern Mining algorithm [Goethals, 2003] to extract discriminative patterns according to neighbourhood size N;
6 Identify functions in the subtree that are executed more frequently in fail test cases than pass test cases;
7 **Analysis phase**;
8 Apply ranking function, f of the neighbourhood size N using

$$P(f) = \frac{\textit{support of f in all fail test cases}}{\textit{support of f in all test cases}}$$

The next phase is the *Filtering phase*. In this phase, the frequent pattern mining algorithm of Goethals [2003] is used to identify the subtrees that are frequently executed in the pass and fail test cases (discriminative patterns). The input to this phase is the neighbourhood size N parameter. The neighbourhood size N refers to the N number of neighbour functions connected to a particular function in a subtree. The neighbourhood size N connected to a function in the subtree might be related to the bug if the function in the subtree is not the bug. Based on this parameter, functions in the subtrees that are executed most of the time in the fail test cases but not in the pass test cases are obtained.

Finally, in the *Analysis phase*, functions are ranked according to their likelihood of being buggy. $P(f)$ refers to the probability of the support of the buggy function that appears in the fail test cases divided by the support of the buggy function that appears in all the test cases (Step 8 of Algorithm 25). A list of the functions of the program are ranked and presented to the programmer to locate the buggy functions of the program.

Di Fatta et al. [2006] measure the effectiveness of their approach using the percentages

of the program code (functions) that the programmer does not need to examine in order to locate the buggy function divided by the number of functions in the program. By using Siemens Test Suite, their approach outperforms other approaches such as the nearest neighbour [Renieres and Reiss, 2003], cause-transition [Zeller, 2002], and SOBER [Liu et al., 2005] approaches. Di Fatta et al. [2006] are able to locate 22% of the buggy functions in the Siemens Test Suite with the frequent patterns of neighbourhood size 2.

Jiang et al. [2005] propose using one of the machine learning techniques – Support Vector Machine (SVM) [Steinwart and Christmann, 2008] – to locate bugs using predicate-based spectra coverage. They combine the latter technique with the cause-effect chain approach [Zeller, 2002] to narrow down the search for the predicates of a program that cause the bug. Initially, they use the CBI system [Liblit et al., 2005] in order to instrument and gather the predicate-based spectra coverage of the program. They rely on the *oracle* of the program code to determine the correctness of the test cases [Jones and Harrold, 2005, Abreu et al., 2006, Wong et al., 2007].

Pseudocode of the method just described is presented in Algorithm 26. The inputs to the program are the predicate-based instrumented program code and the predicate-based spectra coverage (in a binary form, which is similar to the Table 2.2). There are three phases involved in this algorithm. The initial phase is *Classification*, where the Support Vector Machine (SVM) [Steinwart and Christmann, 2008] technique is applied to classify the predicates that are likely to be the bug. Jiang et al. [2005] use both linear and radial basis functions [Walczak and Massart, 1996] to determine the hyperplanes of the SVM. The predicates that are likely to be buggy are extracted from the hyperplanes. Scores of the predicates are then internally assigned using a random forest algorithm [Breiman, 2001]. The predicate with the highest score is chosen as the bug-related predicate. This process of predicate selections is known as feature selection in the machine learning community, in which only the predicates (features) deemed important are selected.

In the second phase (*Clustering*), the selected predicates are clustered according to the distribution of the predicates in the test execution coverage. The latter approach has been used in Liu et al. [2005]. The differences in the distribution of predicates in the test execution coverage are computed for all pairs of predicates. A small threshold value, ϵ, has to be specified by the programmer. If the difference in the distribution of a pair of predicates in the test execution coverage is less than the ϵ, the pair of predicates are deemed similar. Otherwise, they do not belong to the similar cluster. Clusters consist of similar distributions of predicates in the test execution coverage.

In the final phase (*Cause-effect chain*), the relationships between predicates in each cluster are examined using the approach of Zeller [2002]. In each cluster, it is determined whether the predicates are bug-related. Control flow graph (CFG) of the program is gen-

Algorithm 26: Algorithm of Automatic Isolation of Cause-Effect Chains with Machine Learning Approach [Jiang and Su, 2005]

```
Input: predicate-based instrumented program, predicate-based spectra coverage of the
       program
Output: chain of predicates that are likely to be buggy
Classification;
Apply predicate-based spectra coverage as the input for SVM algorithm [Steinwart
and Christmann, 2008];
Assign score to the predicates of the program using random forest [Breiman, 2001];
Perform feature selection to choose top predicates that are most likely the bug;
Clustering;
foreach pair of predicates chosen in feature selection do
    Gather the differences of the distribution of pairs of predicates in all test execution
    coverage [Liu et al., 2005];
    if differences of distribution of the predicates in the test execution coverage < ϵ
    then
        Predicates belong to the similar cluster;
    end
end
Cause-effect chain;
foreach cluster do
    if any predicates in the cluster related to the bug then
        Use Control Flow graph (CFG) to find paths that are related to the predicate;
        Form a chain of predicates;
    end
end
Present the chain of predicates for all the clusters to the programmer;
```

erated to relate the predicates in the cluster with other predicates of the program. A chain of bug-related predicates are formed in each cluster.

Finally, the chain of predicates for each cluster are presented to the programmer for the purpose of bug localization. Jiang et al. [2005] are able to locate 74 bugs of the Siemens Test Suite by examining not more than 10% of the program code. For the Rhythmbox v0.6.4 dataset, they are able to locate 5 bugs after examining 1000 lines out of 56 484 lines of code. This study is the first attempt at combining machine learning approaches with the chain-effect approach [Zeller, 2002] to discover the relationship between bug-related predicates. Jiang et al. [2005] observe that this approach causes performance overhead on larger size programs such as the Rhythmbox dataset.

Liblit et al. introduce sparse random sampling on predicates to locate the likely bugs in a program [Liblit et al., 2003]. The details of their study can be found in Section 3.3. They also propose using machine learning concepts to locate bugs using a regression model, known as Logistic Regression. Pseudocode of this approach is presented in Algorithm 27. The input to this algorithm is the predicate-based spectra coverage (with test cases labelled as Pass or Fail). Predicates are trained using the logistic regression model [Friedman et al., 2001]. A logistic function is produced with β coefficients for each predicate of

the program. The β coefficient of a predicate that is different from other predicates' β coefficient indicates that the predicate is more likely to be buggy.

Algorithm 27: Logistic Regression Algorithm [Liblit et al., 2003]

Input: predicate-based spectra coverage labelled with Pass or Fail
Output: predicates likely to be the bug
1 **Classification**;
2 Predicates are trained using logistic regression model to produce logistic function;
3 Generate the β coefficients of the logistic function for each predicate;
4 Predicate with the different β coefficients as compared to the other β coefficients of predicates is the possible bug;

Zheng et al. previously propose the use of classification and feature selection approaches to narrow down the search for predicates that are likely to be buggy [Zheng et al., 2003]. They use the sampling framework of Liblit et al. [2003] to sample the predicates of the program. However, Zheng et al. [2003] observe that these predicates are not useful indications of the bug of the program. Therefore, they propose to use a clustering approach – a bi-clustering scheme on the predicate-based spectra coverage [Zheng et al., 2006]. They want to cluster and distinguish predicates of the program that are responsible for a particular bug in the program.

Pseudocode of the clustering approach of Zheng et al. [2006] is presented in Algorithm 28. The input to this algorithm is the predicate-based spectra coverage of sampled predicates (frequency counts, which is similarly shown in Table 2.1). This refers to the frequency (number of times) of a particular sampled predicate executed by the pass and fail tests cases. In the first phase, *Inferring Truth Probabilities*, they propose to use a graphical model [Lauritzen, 1996] to infer the truth probability of predicate, $Pred$, in the test executions. For each predicate, they estimate the probability of the predicate being executed and *True* in the respective test cases. The truth probabilities of the predicates are used as the input to the spectral clustering algorithm [Ng et al., 2001]. Using this algorithm, predicates are clustered based on the distances of the truth probabilities between the predicates. In the next phase, votes are assigned to the predicates in each cluster. By assigning votes to the predicates, Zheng et al. [2006] want to determine the particular predicate, $Pred$, that is most likely to be the bug in each cluster. Q_{Pred}, which is the quality of a predicate is defined in the algorithm. Weight is assigned to each predicate of the program using Q_{Pred}. More weights is assigned to a predicate if the predicate is executed in more fail test cases than the pass test cases. The contribution of the predicate, $Pred$, is computed by taking account of the probability of the predicate, $Pred$, being executed in the test cases. Based on the above properties, a vote is assigned to predicate, $Pred$, for each test case. The vote of predicate, $Pred$, in each cluster is then aggregated across all the test cases. The fine details of these formulas can be found in the study of Zheng et al.

[2006]. Predicate, $Pred$, with the highest total votes in each cluster is identified as one of the bugs of the program.

Algorithm 28: Algorithm of Simultaneous Identification of Multiple Bugs Approach [Zheng et al., 2006]

```
   Input: predicate-based spectra coverage of sampled predicates
   Output: predicate with the highest votes for each cluster
   // Q_Pred as quality of each predicate, Pred
 1 Inferring Truth Probabilities;
 2 Sampled predicates are inferred with truth probabilities using graphical model;
 3 Generate clusters of predicates using spectral clustering algorithm [Ng et al., 2001];
 4 Collective Voting approach using bi-clustering scheme;
 5 foreach cluster of predicates formed do
 6     foreach predicate, Pred, in the cluster do
 7         foreach test case do
 8             Compute the Q_Pred;
 9             Contribution of predicate, Pred to respective test case is computed;
10             Aggregate vote assigned by the test case for predicate, Pred;
11         end
12     end
13     Predicate, Pred with the highest number of total votes is chosen as the most
       likely bug in the cluster;
14 end
```

Zheng et al. [2006] evaluate the effectiveness of their proposed approach on the Siemens Test Suite using PDG [Renieres and Reiss, 2003, Cleve and Zeller, 2005] (see Subsection 4.3.3). They observe that their proposed approach is able to locate an additional 70 and 65 bugs of the SOBER [Liu et al., 2005] and CBI [Liblit et al., 2003] approaches respectively by examining more than 7% of the program code.

3.7 Summary

In this chapter, a survey on software fault localization techniques has been presented. These techniques are broadly divided into several studies that use different approaches to locate the bug(s) of a program, namely: slicing and dicing, spectra-based, state-based, test reduction, and combining the spectra-based and machine learning approaches. We have discussed how these approaches are able to locate bugs in a program and have also empirically described the performance of these approaches.

4

Performance Measures

4.1 Introduction

In this chapter, we introduce general principles of performance measures for bug localization; in particular we consider issues such as the granularity of the program (e.g which statements of program to consider), ties between statements having similar metric values, undefined metric values due to division by zero in the evaluation of spectra metrics, and rounding errors in determining the rank of buggy statements. We discuss several performance measures commonly used in the debugging area, and propose several new performance measures. We also detail the benchmarks and the statistical validation used in the thesis.

4.2 Principles of Performance Measures

We have discussed different granularities, for example, statement-based, block-based, predicate-based, function-based, and et al.in Chapter 2. We refer to these granularities as a coverage type. Several issues are to be considered in the performance measure for bug localization. In this section, we briefly discuss these issues and show how they can influence performance measures. Statement-based is used as the context throughout the thesis.

Granularity

One of the principles to consider in evaluating performance measures is the granularities involved with respect to the coverage types.

For statement coverage, there are different lines of code to be considered for the evaluation of respective performance measures. One can consider all the lines of code in the

program. Alternatively, one also consider only the lines of code executed by at least one test case in the test suite.

In several previous studies [Jones and Harrold, 2005, Abreu et al., 2006, Wong et al., 2010], `gcov` (part of the `gcc` compiler suite) has been used for program instrumentation. `gcov` reports execution statistics for each line of source code, including lines that are empty or contain only preprocessor directives such as `#define`, comments, braces, and etc. In our study, we remove all the lines of code (statements) which are not executed by any test cases. These removed statements are the commented and blank lines. Jones et al. remove blank lines, comments, function and variable declarations, and function prototypes from the program code [Jones and Harrold, 2005]. They also treat statements (usually conditional expressions) that span more than one line of code (multi-line statements) as one line of code. In a recent study, Wong et al. also consider only the lines of code executed at least by a test case [Wong et al., 2010]. In Chapter 5, we report the performance measure figures of our proposed bug localization approach using all the lines of code and using only the lines of code which are executed.

Other related studies that report the performance measure figures do not explicitly mention which lines of code they consider [Renieres and Reiss, 2003, Liblit et al., 2005]. This suggests that they may have considered all the lines of code. For example, Renieres et al. use program dependence graphs (PDG) as a performance measure to evaluate their proposed approach [Renieres and Reiss, 2003]. The PDG is generated statically (static analysis), and the selected set of nodes might include nodes of program code that is not executed (the details of this measure are presented in Subsection 4.3.3).

The number of lines of code considered influences the bug localization performance for any performance measure used. Assume we are considering two programs; a program that consists of 10 lines of code which are executed at least once by a test case, and a program that consists of 1 million lines of code, whether or not they are executed by test cases. We make an assumption that we use a particular spectra metric, and that the buggy statement is located at the third position for both programs. Using the performance measure of rank percentages (detailed in Subsection 4.3.1), the program with 1 million lines of code will yield extremely good performance (very small rank percentages) as compared to the program with 10 lines of code.

Apart from program statements, other studies use basic blocks (block-based spectra coverage) to evaluate their proposed approach to improve bug localization performance [Abreu et al., 2006, Abreu et al., 2007]. One of the advantages of using block-based spectra coverage is that a particular basic block coverage of a program represents the set of statements covered by the block. Using program statements may cause ties of $MetValue$ among statements within the same block of a program. The latter case would

not occur using block-based spectra coverage. The current version of `gcov` supports block coverage. However, it also produces other information e.g description of branch coverage at the end of each block that needs to be filtered. For this reason, some previous studies develop their own tools, such as ATAC [Horgan and London, 1992] and Front parser [Abreu et al., 2006, Abreu et al., 2007]. Lucia et al. have manually instrumented the block coverage of programs in their recent study [Lucia et al., 2010].

Several studies have proposed to use predicates in the program code for bug localization [Liblit et al., 2005, Liu et al., 2005, Jiang and Su, 2005, Chilimbi et al., 2009]. Each predicate is associated with a single program point. Examples of useful predicates are conditions of `if` statements, and whether the `return` expressions of functions are positive, negative, or zero. During the execution of a test case, data can be gathered on the predicates that are *observed* (execution has reached that program point) and of those which are the ones that were *True* (at least once). The details can be referred to Figure 2.1 of Chapter 2. Liblit et al. developed the Cooperative Bug Isolation (CBI) system [Liblit, 2004, Liblit et al., 2005] to instrument the predicates of program code. The advantage of the predicate-based system is that predicates unrelated to control flow can also be introduced; for example, the `return` value of a function. Return values often indicate whether a function has exited successfully. A bug in the function could potentially terminate the program without returning values. The bug can be located using this type of predicate.

The above principles are important when using performance measures to evaluate proposed approaches to locate bugs. For the thesis, we are particularly interested in the statement-based coverage type. In our evaluation of performance measures, we consider only the lines of code or statements that are executed.

Ties

In the ranking of program statements with spectra metrics, there are possibilities of having a similar metric value ($MetValue$) for more than one statement of the program. There are several approaches that have been proposed to handle ties in the ranking of program statements. These approaches are known as *High*, *Low*, and *Mid* measures. Some previous studies have used these different measures to report the effectiveness of their proposed bug localization approaches. *High* refers to the top most position of the statement that shares the same metric value as the other statement(s). The *Low* measure refers to the bottom most position of the statement that shares the same metric value as the other statement(s). Several studies use the *Low* measure [Jones and Harrold, 2005, Abreu et al., 2006, Santelices et al., 2009]. The *High* and *Low* measures are also known as *Best case* and *Worst case* respectively [Wong et al., 2007, Wong et al., 2010]. The *Mid* measure

refers to the average of the position of the statements that share the same metric value. Some of the earlier studies use the *Mid* measure [Abreu et al., 2007, Ali et al., 2009] to report the performance of their proposed bug localization approaches. The *Mid* measure is also referred to as the *middle line measure* by Ali et al. [2009].

We illustrate these approaches with a simple program used by Jones et al. [2002]. Using the program shown in Table 2.4, we observe the *mid* program which consists of 13 statements. Assuming that the bug is located in Statement 2, we observe that the a_{ep} and a_{ef} for Statements 1, 2, 3, and 13 are the same (a_{ep}=5 and a_{ef}=1). If we evaluate with any of the better performing spectra metrics found in Chapter 5 (O^p, Jaccard, and Tarantula metrics, to name a few), Statement 7 and Statement 6 are ranked the first and the second position respectively. By using any spectra metrics, we observe that Statements 1, 2, 3, and 13 have the same $MetValue$. For the *High* measure, the ranking position of the bug would be 3 since Statement 2 (which contains the bug) could optimistically be ranked right after Statement 7 and Statement 6 in the list, separately from the other three statements. For the *Low* measure, the ranking position of the bug would be 6, since Statement 2 could be ranked bottom in the list, right after Statements 7, 6, 1, 3, and 13. For the *Mid* measure, the ranking position would be 4.5, since there are 4 statements that share the same $MetValue$ in the ranking. The average position among the four statements is taken in this case.

In the case of having ties in multiple-bug programs, we only consider the ties of the highest ranked buggy statement found by the programmer. Any tie that exists between the highest ranked buggy statement and other non-buggy statement(s) are considered as part of the ranking for the *High*, *Low*, and *Mid* measures.

Reporting both *High* and *Low* measures for bug localization performance is not so useful because it does not give a single number, and makes the comparison between spectra metrics harder. Using the *High* or *Low* measure itself is misleading for some metrics (for example, the Russell metric). We defer an explanation of this to Subsection 5.11.1 of Chapter 5 where we evaluate these different measures on the Siemens Test Suite benchmark. It is more rational to use the *Mid* measure, as it takes into account the average of all the possible positions of the statements having the same $MetValue$ in the ranking. The *Mid* measure does not give any extreme bug localization performance figures, as compared to using the *High* and *Low* measures. This measure is therefore more realistic when compared to using the *High* and *Low* measures. Reporting a single number such as the *Mid* measure also helps to compare the bug localization performance between different spectra metrics.

Undefined Metric Values due to Division by Zero

We also have to consider ways to handle the issue of having undefined metric values caused by division by zero, in the ranking of program statements. Previous studies have not addressed this issue [Abreu et al., 2006, Wong et al., 2007, Wong et al., 2010, Debroy et al., 2010].

When it comes to ranking program statements, there is a possibility of the denominator of respective spectra metrics having zero. We could handle this scenario in three different ways.

1. Return a large metric value
2. Assign zero to the statement
3. Use ϵ on the denominator

The first solution when the denominator has zero is to return a suitably large metric value. For example, when using the Tarantula metric to evaluate the metric value of program statements, if the denominator of a statement is zero, rather than returning an undefined value, we could use a larger value such as the number of tests plus 1, which is larger than any value which can be returned with a non-zero denominator.

Another possibility to handle undefined metric values (due to the division by 0) is to assign zero to a statement when the denominator of that statement is zero. For example, take CBI Log (refer to the metric in Table 2.3). The log in the CBI Log metric will give undefined value when the denominator is 0. Liblit et al. [2005] assigned zero to the predicates when the denominator is zero in the CBI Log and CBI Sqrt metrics. We handle the similar way for both of these metrics in this thesis.

Definition 10 (Ample metric)**.**

$$Ample = \left| \frac{a_{ef}}{totF} - \frac{a_{ep}}{totP} \right|$$

The third solution proposed to handle the denominator being zero is to add a suitably small ϵ to the denominator. There is no issue when applying ϵ on the denominator for most of the spectra metrics with the exception of the Ample metric. We observe differences in bug localization performance when we apply ϵ on the denominator for the Ample metric. The Ample metric (Definition 10) consists of two terms, the proportion of fail test cases and the proportion of pass test cases. Of the two terms, the smaller term is subtracted from the larger term.

In Table 5.1 of Chapter 5, using the top-rank-bug score detailed in Subsection 4.4.2 on the $ITE2_8$ model program, we observe that the Ample metric (without ϵ) yields a score of 46.36% for 100 tests. Using the same number of tests, Ample metric with ϵ yields a score of 6.39%. The reason for this different observation is due to having ties for statements S3 and S4 of the $ITE2_8$ model program when using the Ample metric without ϵ.

We illustrate the above observation with a simple example. Assume we have a statement, S1, with a_{ef}=1, a_{ep}=0, a_{nf}=0, and a_{np}=4. Another statement, S2, has a_{ef}=0, a_{ep}=4, a_{nf}=1, and a_{np}=0. If we use the Ample metric with the ϵ value, S1 and S2 would be $|\frac{1}{1+\epsilon}\text{-}\frac{0}{4+\epsilon}|$ and $|\frac{0}{1+\epsilon}\text{-}\frac{4}{4+\epsilon}|$ respectively. S2 would have a higher $MetValue$ since S2 has a numerator of 4, whereas S1 has a 1 as the numerator. By using the Ample metric without the ϵ value, S1 and S2 would have $|\frac{1}{1} - \frac{0}{4}|$ and $|\frac{0}{1} - \frac{4}{4}|$ respectively. In this case, S1 and S2 would have the same $MetValue$ of 1. This example shows that using the Ample metric without the ϵ value introduces ties for statements. This affects the buggy statement position in the ranking especially for small size programs such as the $ITE2_8$ model program. We can observe the bug localization performance for the Ample metric is at the bottom of the Table 5.1. However, the bug localization performance is not affected by introducing the ϵ in the denominator in the metric on larger program size of the benchmarks (Siemens Test Suite, the subset of the Unix Test Suite, Space, and Concordance).

In the thesis, we consider both cases; assigning zero for some metrics, namely CBI Log and CBI Sqrt, when the denominator is zero, and assigning ϵ to most of the metrics. We do not assign any ϵ for the Ample metric.

For metrics that are found to be equivalent in the ranking using a monotonically increasing function (Lemma 5.2.1), there are slight differences in the metric values when we apply ϵ to these metrics. These metrics are detailed in Subsection 5.2.1. Take for example, the Simple Matching and Manhattan metrics (refer to Table 2.3). When we evaluate these metrics on a program statement, a rounding error is observed in the metric value of this statement on both metrics. However, the rounding error does not affect the ranking of the buggy statement for most of these metrics.

Rounding Errors

We also observe rounding errors in determining the ranking of the buggy statement(s). In the ranking of program statements, the metric value $MetValue$ of each statement (evaluated with most of the spectra metrics) is always a real number. Floating point values are just an approximation of the real numbers in this case.

For the rank percentages measure (refer to this performance measure in Subsection 4.3.1), we use a small ϵ value as the relative error to compare the $MetValue$ of the program statements. For example, using a spectra metric, there is a possibility of having a

group of statements with a $MetValue$ of 0.5 and another group of statements in the program with a $MetValue$ of 0.49999. We make an assumption that the buggy statement is part of the group of statements with a $MetValue$ of 0.5. If we use a slightly larger ϵ value than 0.00001 for the relative error in the comparison of program statements' $MetValue$, the group of statements with $MetValue$ of 0.49999 will be rounded to 0.5 (rounding error). All of these statements in both groups will have a similar $MetValue$. Using the *Mid* measure, the ranking of the buggy statement would be the average bug position of all of the statements in both of the groups. If we use a smaller ϵ value than 0.00001 for the relative error in the comparison of program statements' $MetValue$, the ranking of the buggy statement would be different. The group of statements with a $MetValue$ of 0.49999 will not be rounded to 0.5. Using the *Mid* measure, the ranking of the buggy statements would be the average bug position of all the statements in the group of statements with a $MetValue$ of 0.5. Regardless of the ϵ value that we choose for the relative error, there is a possibility that the ranking of the buggy statement might be slightly different due to the ϵ value. A tiny difference in the metric value, $MetValue$, affects the ranking of potentially buggy statements and the reported bug localization performance in our evaluation.

4.3 Existing Performance Measures

Several approaches have been proposed in previous studies to improve bug localization performance [Renieres and Reiss, 2003, Cleve and Zeller, 2005, Jones and Harrold, 2005, Abreu et al., 2006, Wong et al., 2007, Wong et al., 2010]. These studies use performance measures to report and compare the effectiveness of their proposed bug localization approaches. Usually, these performance measures normalise with respect to the program size and the types of coverage deployed. We compare the performance of bug localization approaches based on the relative metric ordering of the spectra metrics. We discuss three of the most common performance measures used by most of the existing studies. There are rank percentages, successful diagnosis of bugs, and program dependence graphs (PDG).

4.3.1 Rank Percentages

Jones et al. propose to measure the proportion of the ranked program statements of the program code that the programmer need not examine before the bug is found [Jones and Harrold, 2005], whereas Abreu et al. propose to measure the proportion of the ranked program blocks of the program code (block-based spectra coverage) that the programmer needs to examine before the buggy block of the program is found [Abreu et al., 2006].

In their study, the position of the buggy block of the program is known as q_d. We use similar performance measure to the latter except that it is statement-based. We refer to this performance measure as rank percentages (Definition 11). We introduce the term rank percentages because the figures are generated from the rank of program statements and represented in percentages. It is essentially similar to the measure proposed in previous studies such as Yu et al., Wong et al., and Xie et al. [Yu et al., 2008, Wong et al., 2010, Xie et al., 2010]. They have introduced different namings but still refer to the same performance measure. For example, Yu et al. [Yu et al., 2008] considers $Expense$ to measure the proportion of the ranked program statements of the program code that the programmer needs to examine before the buggy statement of the program is found. Wong et al. [Wong et al., 2010] and Xie et al. [Xie et al., 2010] have also introduced similar measures to $Expense$, and they are known as $EXAM$ and p_r respectively.

Definition 11 (Definition of rank percentages)**.**

$$\frac{\textit{Number of statement(s) to be examined in order to find the bug}}{\textit{total number of statements}} * 100$$

Definition 11 refers to the percentage of the program code (program statements) that have to be examined in order to find the bug. Note that in this definition, the total number of statements can be considered differently (discussed in Section 4.2). Using different total number of statements affects the figures reported for rank percentages. In the thesis, we particularly consider the number of program statements that are executed at least once in the test cases which are similar to the previous studies [Abreu et al., 2006, Yu et al., 2008, Wong et al., 2010, Xie et al., 2010].

Rank percentages are calculated with reference to a complete ranking of all the statements in the program. Smaller rank percentages indicate that the programmer examines a smaller portion of the program code before the bug is found. Bug localization performance is deemed good when we have smaller rank percentages. Other studies propose slightly different variants from our definition of rank percentages [Jones and Harrold, 2005, Abreu et al., 2007]. One of the variants proposed in the existing studies is the complement of rank percentages, which is denoted as q_d. This refers to the number of statement(s) or block(s) not needed to be examined in the program code before the bug is found. The higher the q_d [Abreu et al., 2007], the better the bug localization performance, since the programmer needs to examine lesser program code to locate the bug.

In practice, the number of bugs in a program is not known to the programmer. The programmer examines the bug which has been found first and fixes that bug before proceeding to examine the next bug in the program. Therefore, for multiple-bug programs, we only consider the highest ranked buggy statement found by the programmer as the bug

for the rank percentages. Across multiple programs in a test suite, we report the average rank percentages. Abreu et al. also consider the latter when they report bug localization performance for programs in the Siemens Test Suite [Abreu et al., 2006]. Average rank percentages is used throughout the thesis to report the performance of proposed bug localization approaches.

4.3.2 Successful Diagnosis of Bugs

Several studies measure the effectiveness of their proposed bug localization approaches based on the successful diagnosis of bugs [Renieres and Reiss, 2003, Jones and Harrold, 2005, Hao et al., 2005, Jiang and Su, 2005, Liblit et al., 2005, Liu et al., 2005]. These studies report the percentage of programs where the bug has been successfully found for a given percentile of the program code examined by the programmer. Similarly, Cleve et al. measure the percentage of bugs successfully found within different percentages of nodes in the PDG examined by the programmer [Cleve and Zeller, 2005]. We name the successful diagnosis of bugs measure as $SucDiag$, and use this term throughout the thesis. The $SucDiag$ measure considers a complete ranking of all the statements of the program.

Due to possible ties in the ranking of buggy program statements (discussed in Section 4.2), different steps have to be applied to compute the $SucDiag$ measure. We handle the $SucDiag$ measure for two cases; the case where the buggy statement does not has any ties with other non-buggy statement(s) and the case where the buggy statement has ties with other non-buggy statement(s). In the first case, if we choose to examine at most 10% of the statements in a program and the top ranked 10% of the statements include the bug, we assign a value of 1. This indicates the bug is successfully found when the programmer examines at most 10% of the statements in the program. For example, if we have 70 of the 120 programs in the test suite with the above condition, that would be $70/120 \times 100 = 58.34\%$.

For the second case, when the buggy statement is tied in the ranking with other non-buggy statement(s) (having a similar metric value), we assume the bug can be found anywhere in that range with uniform probability. Suppose the top 10% of statements are all ranked equally including the bug. Successful diagnosis, $SucDiag$, is assured by examining 10% of the program. However, if only 8% of the program is examined, successful diagnosis is likely but not assured. We give a value of 0.8 to indicate the probability. In this case, the $SucDiag$ of a typical test suite for the range of 10% would be the sum of these probabilities, divided by the number of programs, times 100%.

Instead of having only 10% of statements to be examined to locate the bug, multiple percentile ranges (percentages of program code to be examined) can be defined. For example, we define seven percentile ranges (see Table 5.13 and Table 5.14) in order to

make a fair comparison of our proposed bug localization approach with previous studies [Cleve and Zeller, 2005, Jiang and Su, 2005, Liblit et al., 2005, Liu et al., 2005]. A higher $SucDiag$ for given percentile ranges indicates more programs where the bug is found within the percentile ranges in the program. For comparing bug localization performance between spectra metrics, we are particularly interested in smaller percentiles as the programmer could examine smaller percentages of the program code to locate the bug. However, the disadvantage of using this performance measure is having multiple percentile ranges. Using multiple percentile ranges is hard to compare $SucDiag$ for different spectra metrics.

4.3.3 Program Dependence Graphs (PDG)

Renieres et al. propose the use of program dependence graphs (PDG) in order to measure the bug localization performance of their nearest neighbour approach [Renieres and Reiss, 2003]. The details of the nearest neighbour approach can be referred to in Chapter 2. Cleve et al. and Liu et al. have also used PDG as the performance measure of the bug localization performance of their proposed approach [Cleve and Zeller, 2005, Liu et al., 2005].

To use this measure, the PDG for the program code has to be generated [Renieres and Reiss, 2003] using a tool such as CodeSurfer [Teitelbaum et al., 2001]. Every expression of the program code (for instance, every statement or block of the program) is treated as one of the nodes of the PDG. Using the respective proposed bug localization approaches [Renieres and Reiss, 2003, Cleve and Zeller, 2005, Liu et al., 2005], a set consists of nodes (statements or blocks) likely to be buggy is produced. The PDG of the program code is traversed to examine any dependency neighbourhoods of each *node* in the set until the programmer recognises a buggy node. The traversal of these nodes is performed using breadth-first search [Horowitz et al., 1995]. Each set of nodes that has a directed path from the *node* to the buggy node is computed. Nodes that have been traversed from the respective *node* in the set to the buggy node are gathered; this is known as the *node set*. A measure (known as Score in their study) is established [Renieres and Reiss, 2003] and is represented by the following definition (Definition 12).

Definition 12 (Definition of measure using PDG defined by [Renieres and Reiss, 2003])**.**

$$1 - \frac{\textit{number of node(s) in the smallest node set}}{\textit{total nodes in overall PDG}}$$

Definition 12 refers to the proportion of the number of nodes not needed to be examined in the PDG before reaching the buggy node, divided by the total number of nodes in the overall PDG. The number of nodes refers to the nodes in the smallest node set. If the

node in the smallest node set is the buggy node, and this is the only node in the node set, the search will stop immediately; the measure returned would be 1. The measure returned is smaller and closer to 0, the more nodes there are in the smallest node set.

Liu et al. propose a similar methodology for measuring performance by traversing the PDG in a breadth-first-search until the buggy node is found [Liu et al., 2005]. However, they propose the complement of Definition 12, which is referred to as $T - score$ in their study. The $T - score$ measure refers to the proportion of nodes (statements or blocks) that need to be examined in the PDG before reaching the buggy node, with respect to the total number of nodes in the overall PDG.

By using this measure, the program dependence graph (PDG) of the program code has to be generated. The *node set* for each *node* that is likely to be buggy in the PDG has to be determined. Unlike previous performance measures such as rank percentages and $SucDiag$, this measure does not assume a complete ranking of all the statements in the program. It only considers the smallest node set that reaches the buggy node. Therefore, this measure is not comparable to the other performance measures such as rank percentages and $SucDiag$.

4.4 Proposed Performance Measures

We propose several new performance measures, and evaluate these performance measures on our benchmarks (see Section 5.11 of Chapter 5). These measures are listed as below.

1. Median rank percentages.
2. Top-rank-bug score.
3. Relative score.

4.4.1 Median Rank Percentages

By using the average rank percentages measure in Subsection 4.3.1, the bug localization performance for one program might be very different from that for other programs. We propose the use of *median* rank percentages to avoid the case of a program that performs very differently (an outlier) from other programs. We also use the First Quartile (25th percentile) and Third Quartile (75th percentile) [Dodd, 1938]. This enables us to understand the spread of the bug localization performance of the test suite programs evaluated with each spectra metric.

The median rank percentages measure is essentially the inverse of the $SucDiag$. Instead of analysing different percentages of the program code (up to 50% of the program

code to be examined) using $SucDiag$, the programmer can gain an insight into bug localization performance by just analysing the median (50th percentile).

Some of the metrics might show a wide spread due to different bug localization performances among the different versions of each program. Since the median is a single number, it is easy to compare different spectra metrics. The median rank percentages is a useful and robust measure as it does not consider the bug localization performance for a program which is very different from that of other programs.

4.4.2 Top-rank-bug Score

We propose another measure known as the *top-rank-bug* score. This measure examines the proportion of programs having their bug ranked top, and treats equally ranked statements in a reasonable way. The top-rank-bug score for a program evaluated with a metric is 100% if the buggy statement is ranked highest and no tie exists with the other program statements. If the buggy statement is ranked equal-highest with k other statements, the top-rank-bug score is $\frac{1}{k} * 100\%$. If the bug is not located at the top of the ranking list, the top-rank-bug score is 0%.

This performance measure is extremely simple and might not be sensible if the program is large and has more than one bug. We use this measure to evaluate and analyse the simple model program ($ITE2_8$), which consists of only four statements, in Chapter 5. Top-rank-bug score is a useful measure to indicate the proportion of the programs in which the bug is ranked top.

4.4.3 Relative Score

The more fail test cases (a_{ef}) and the fewer pass test cases (a_{ep}) that execute a statement, the more likely the statement is buggy. Any sensible metric would rank a statement higher than the other statement in the ranking list if the statement has larger a_{ef} and a_{np} values than the other statement (a larger a_{np} is equivalent to having a smaller a_{ep}). Regardless of any spectra metric used, it is possible for non-buggy statements that rank higher than a buggy statement to have a larger a_{ef} and a_{np} than the buggy statement. It is also possible for non-buggy statements to share identical a_{ij} values with the buggy statement. These statements would have similar metric values and cause bug localization to perform poorly regardless of the metric used. Therefore, no sensible spectra metric could rank the program bug on top in these cases.

We propose the *relative score* measure to evaluate the best possible bug localization performance a spectra metric could achieve, if such cases were ignored. Based on the ignored cases, we examine two *conditions* on non-buggy statements that rank higher than

the buggy statement in the ranking list. For the first condition, we examine whether the a_{ef} values of these non-buggy statements are less than the a_{ef} value of the buggy statement. In the second condition, we examine whether the a_{ep} values of these non-buggy statements are larger than the a_{ep} value of the buggy statement. These conditions allow us to observe how closely a metric can achieve best bug localization performance as compared to other metrics.

We describe the pseudocode for *relative score* in Algorithm 29. The inputs to this algorithm are the program statements evaluated with the given spectra metric (sorted in descending order) and buggy statement *S*. Initially, the buggy statement position has to be identified from the ranked program statements. In the ranking of the buggy statement, there is a possibility of other non-buggy statements sharing the same metric value with the buggy statement. Therefore, we have two inputs, fp and lp in Algorithm 29 which refer to the first and the last position of the statements sharing the same metric value with the buggy statement respectively. The fp and lp are identical if there is no other non-buggy statement(s) sharing the same metric value with the buggy statement. In this algorithm, if the buggy statement is tied with other non-buggy statements, the first and last position (fp and lp) in the ranking have to be examined. For multiple-bug programs, we always consider the position of the bug that is first found in the ranking.

If the bug is ranked at the top, and no ties exist with non-buggy statements, the algorithm skips the two *while()* loops and returns the relative score of 1 (Definition 13). Otherwise, the first while loop is used to examine the program statements that rank higher than the buggy statement (first position, fp). The two *conditions* mentioned are used in this loop to determine the $nonbug$ counter variable (which is part of the relative score). This step is repeated until the first position, fp, has been reached.

Once the algorithm has reached the first position, fp, it examines any tie (that is, identical metric value of the first position, fp, with other statements) in the ranking. If there is a tie, the algorithm needs to examine the second while loop. This second while loop works similarly to the first while loop. The two *conditions* in Step 7 are used to determine part of the relative score using the $nonbugties$ counter variable. The second while loop will be skipped if a tie does not exist between the first position, fp, and the consecutive statements in the ranking.

Definition 13 (Relative Score, Rel_p)**.**

$$Rel_P = (1 - \frac{nonbug + \frac{nonbugties}{2}}{\textit{\# of statements in the program}}) * 100$$

Algorithm 29: Algorithm of Relative Score

Input: ranked program statement s according to $MetValue$ (in descending order), buggy statement *S*, first position, fp and last position, lp of the program
Output: relative score of the program, Rel_p

```
while top ranked statement s has not reached fp do
    if (a_ef(s) < a_ef(S) || a_ep(s) > a_ep(S)) then
        nonbug++ // nonbug as variable counter
    end
end
while (MetValue(s) == MetValue(S) && the statement s has not reached lp) do
    if (a_ef(s) < a_ef(S) || a_ep(s) > a_ep(S)) then
        nonbugties++ // nonbugties as variable counter
    end
end
Return relative score Rel_p (Definition 13) ;
```

The relative score, Rel_p, defines the best possible bug localization performance that a metric can achieve for a typical program (Definition 13). This score is quite similar to the rank percentages (Subsection 4.3.1). The ties ($nonbugties$) with the buggy statement handled in this score is similar to that obtained using *Mid* measure. However, the relative score is the reverse of the rank percentages. A larger relative score indicates that the metric could achieve better bug localization performance; that is, the buggy statement of the program can be located faster. By using this measure, we can observe the best possible bug localization performance a spectra metric can achieve.

4.5 Empirical Datasets and Validation

We evaluate several performance measures on benchmarks in the thesis. Table 4.1 describes the breakdown of the programs in the benchmarks with the number of programs of different number of bugs. The table also details the number of lines of code (LOC), total number of test cases, and the number of pass and fail test cases of the programs. In this table, most of the programs belong to either Siemens Test Suite or Unix Test Suite. We use the term test suite to refer to Siemens and Unix datasets as previous studies have been using this terminology [Renieres and Reiss, 2003, Tallam and Gupta, 2005].

The first seven programs in Table 4.1 are from the Siemens Test Suite [Do et al., 2005]. This test suite is a widely used benchmark for bug localization [Renieres and Reiss, 2003, Pytlik et al., 2003, Jones and Harrold, 2005, Liu et al., 2005, Abreu et al., 2006, Abreu et al., 2007, Wong et al., 2007]. $print\_tokens$ and $print\_tokens2$ denote lexical analyser program code. tot_info and $replace$ are information measure and pattern

Table 4.1: Description of the Siemens Test Suite, the subset of the Unix Test Suite, Space, and Concordance

Program	1 Bug	2 Bugs	3 Bugs	LOC	# Test Cases	# Pass	# Fail
$print\_tokens$	6	—	—	563	4130	4050	80
$print\_tokens2$	10	10	44	508	4115	3891	224
$replace$	29	34	16	563	5542	5440	102
$schedule$	8	—	—	410	2650	2555	95
$schedule2$	9	28	131	307	2710	2677	33
$tcas$	37	604	—	173	1608	1570	38
tot_info	23	245	782	406	1052	969	83
Cal	18	115	1475	202	162	90	72
$Checkeq$	18	56	502	102	332	79	253
Col	28	147	2030	308	156	92	64
$Spline$	13	20	174	338	700	648	52
Tr	11	17	90	137	870	707	163
$Uniq$	14	14	114	143	431	273	158
$Space$	15	5	1	9059	13585	11631	1954
$Concordance$	11	—	—	1492	372	353	19

recognition program code. $schedule$ and $schedule2$ denote code for a priority scheduler. $tcas$ is program code for altitude separation. The next six programs in the table are a subset of the Unix Test Suite [Wong et al., 1998]. These programs are various Unix utility programs. Cal is a calendar utility that prints a calendar for a specific year or month. $Checkeq$ reports missing or unbalanced delimiters and .EQ/.EN pairs. Col is a program used to filter reverse paper motions from nroff output for display on a terminal. $Spline$ is a program used to interpolate smooth curves on the basis of given data. Tr is a program to translate characters. $Uniq$ is a utility to report or remove adjacent duplicate lines. The third dataset we use is the Space [Frankl and Iakounenko, 1998, Do et al., 2005] program which provides a user interface to configure an array of antennas. We also use Concordance as a benchmark [Ali et al., 2009] and is a utility for making concordances (word indices) of documents, written by Ralph L. Meyer (`http://hpux.connect.org.uk/hppd/hpux/Misc/conc-0.5/man.html`). This dataset is different from the Siemens Test Suite, the Unix Test Suite subset, and the Space datasets, as the bugs of the program code are not seeded using any tool. Ali et al. identify 13 different types of bugs in the Concordance program code [Ali et al., 2009]. In their study, these bugs are known as naturally occurring bugs and we name it as buggy versions of Concordance. Once they have identified these bugs, Ali et al. [2009] fix the bugs by using preprocessor directives (e.g #if ... #endif). We name the fixed versions of the program code as correct versions of Concordance. The output of the test cases on the

correct versions and the buggy versions of the Concordance program code are compared to determine the correctness of the test cases. The details of the bugs of our benchmarks can be referred to in Appendix H.

These datasets have different program sizes as measured by lines of code (column LOC of Table 4.1). We observe that the Siemens Test Suite and the subset of the Unix Test Suite have the least lines of code executed (on average of 320 lines of code). Therefore, we refer to them as small-size programs. We refer to Concordance as a medium-size program, as it has more lines of code executed (1492 lines of code). We refer to the Space program as a larger-size program, as it has more lines of code executed (9059 lines of code) than the other datasets in Table 4.1. However, the programs in these benchmarks are small in comparison to the size of real-world programs, such as GCC [GCC, 2010a].

In this thesis, program spectra are extracted using `gcov` (part of the `gcc` compiler suite) and Bash scripts. There are six Siemens test suite programs that have runtime errors (segmentation faults) for some tests and `gcov` fails to produce any execution counts; we have obtained data from other researchers for these cases. We encountered four Unix Test Suite subset programs that have runtime errors (segmentation faults). There are also six (of an original 38) Space programs of which we failed to obtain data. We have excluded these programs from the benchmark set.

We distinguish between the number of bugs of each program that we evaluate. In the thesis, we define a single bug as a program with one bug. Not all of the benchmark programs have a single bug that shows unintended behaviour when executed. For example, some programs in the benchmarks have a (single) `#define` that is wrong. The `#define` is never executed; it is only when a statement that uses the macro is executed that unintended behaviour occurs. There are typically several such statements. The terms two-bug and three-bug refer to programs that have two bugs and three bugs respectively. In the thesis, we only consider programs that fail for at least one test case. If the program does not contain any fail test cases, it is not appropriate for dynamic diagnosis as there are no symptoms to diagnose. We define the number of programs that we evaluate in the thesis according to the number of bugs (see Table 4.1), namely, 1 Bug, 2 Bugs, and 3 Bugs columns which refer to one-bug (single bug), two-bug, and three-bug programs respectively.

To create two-bug programs, we picked all the possible single bug (one-bug) versions of the same program of the dataset and combined them as two-bug programs. We did not consider pairs of single bug programs which have identical bugs. The same approach is used to generate three-bug programs, namely by picking all the possible single bug programs and combining them with all the possible two-bug programs. This approach has been used in the previous study of Jones et al. [2007]. We only generated two-bug and

three-bug programs for the Siemens Test Suite and the subset of the Unix Test Suite, as they contain more program versions than the Concordance and Space datasets. Space contains seven versions with more than three bugs. Therefore, we call all the Space versions with more than one bug (13 of them) as *multiple-bug* of Space programs.

For the larger-size program such as Space, we evaluate two different sets of the dataset. In the first set, we use all of the test cases provided in the test suite [Do et al., 2005], which we name as *AllTests*. As the Space test suite consists of a large number of test cases (close to 13 600 test cases), we choose to use a subset of the entire test suite as the second set. A previous study has also instrumented the Space program by choosing to use a subset of the entire test suite [Jones et al., 2007]. For the second set of the Space dataset, we randomly select 10% of the pass and fail test cases of the entire test suite of Space. In order to avoid any bias of specific test selections, we repeat the random test selection 10 times. For each selection, we gather the test cases into a particular `bin`. Therefore, we have 10 bins, each contains a subset of test cases of the Space programs, which we name as *Subset*.

Making the distinction between the number of bugs in a program in our evaluation allows us to gain a better understanding especially for single bug programs. The Siemens Test Suite [Do et al., 2005], which is widely used in the debugging area [Jones and Harrold, 2005, Abreu et al., 2006, Wong et al., 2007], is strongly biased towards single bug programs. Previous studies [Renieres and Reiss, 2003, Wong et al., 2007] report the effectiveness of their bug localization approaches using the entire benchmark, without making any distinction between programs with different numbers of bugs. We observe different bug localization performances on single bug programs and multiple-bug programs in this thesis.

We propose several bug localization approaches that help improve the performance of bug localization in this thesis. We validate the effectiveness of the proposed approaches using a one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973]. We establish the hypothesis of improved bug localization performance using our proposed bug localization approach. We observe the significance of our hypothesis, known as the *p-value*. We choose the commonly used p-value of 0.05 [Bross, 1977] and report the confidence interval [Neyman, 1937] of our hypothesis. A p-value less than 0.05 indicates that the hypothesis of the improved bug localization performance using our proposed bug localization approach is true with the confidence greater than 95%.

4.6 Threats to Validity

In the earlier sections, we discussed several existing performance measures used in debugging, particularly rank percentages (Subsection 4.3.1). The latter is a reasonable performance measure as it helps indicate the programmers' effort to locate bugs on different spectra metrics. In the thesis, most of the figures we reported consider the average of rank percentages for respective programs in each test suite. There might be other possible performance measures which could be used to compare bug localization performance on several spectra metrics, and potentially yield different results (as to which metric programmers should use to locate bugs). One possible performance measure to consider is by taking the pairwise comparison of the rank percentages among two metrics. We illustrate both average rank percentages and pairwise comparison measures in the following example.

Given three programs of 100 lines of code evaluated respectively on two spectra metrics, A and B. The rank percentages for these programs evaluated on metrics A and B are A=$\{2,6,10\}$ and B=$\{1,5,21\}$ respectively. By considering the average rank percentages measure, it would be 6% and 9% on metrics A and B respectively. It has been explained in Subsection 4.3.1 that lesser rank percentages refers to the programmers having less program code to examine in order to locate bugs of the program. As the metric A has smaller average rank percentages than metric B, the former metric would be chosen by the programmers to locate bugs of the program. If we perform pairwise comparison on the rank percentages for each program evaluated with metric A and B, there are two programs where metric B has smaller rank percentages than metric A (1 versus 2, and 5 versus 6). From these pair comparisons, one could conclude that metric B is a better choice for programmer to use to locate bugs as compared to metric A. Pairwise comparison measure could be possibly used in the thesis to provide different perspective to the programmers when choosing metrics to locate bugs. However, there are over 80 spectra metrics in the thesis and it is not feasible to make comparison among the different combinations of pairs of metrics.

We also propose other performance measures in Section 4.4, particularly, median rank percentages. This measure can handle outliers, for example using the previous example, Metric A and Metric B will yield median value of 6 and 5 respectively. We evaluate our metrics using median rank percentages in Table 5.18 (on page 140) and show that the ordering of the metrics, especially the top five metrics in Median column is similar to the ordering of the metrics in Mid column (average rank percentages).

We validate our empirical evaluation of our proposed bug localization approaches using statistical test, t-test [Hollander and Wolfe, 1973]. We validate the significance

of the improvement of bug localization performance using our proposed approaches by comparing the ranking of the buggy statement for each program evaluated. More details of t-test can be referred to Section 4.5.

Another concern is the datasets used (detailed in Section 4.5). Bugs in the Siemens Test Suite are manually seeded by Siemens researchers [SIR, 2010]. For the Unix Test Suite, the bugs are seeded using automated tool [Wong et al., 2010]. The empirical results on these datasets (metrics perform better than the others in bug localization performance) could not be necessarily generalized to real programs. We evaluate Concordance dataset, where the bugs in this program are not seeded by any tool [Ali et al., 2009]. By using this dataset, we show similar ordering of metrics in terms of the effectiveness of bug localization, with the other test suites such as Siemens Test Suite and subset of Unix test suite. However, Concordance has only 11 programs and having more real programs will be future work.

4.7 Summary

In this chapter, we have discussed several general principles to consider when evaluating performance measures. These principles are: the granularity of the program (which statements to consider); ties of statements having the same metric value; undefined metric values due to division by zero; and rounding errors that occur when evaluating with spectra metrics. We detailed the existing performance measures (using rank percentages, successful diagnosis of bugs ($SucDiag$), and program dependence graphs (PDG)) and their disadvantages. This has led us to propose several new performance measures. By using *median rank percentages*, the problem of having different bug localization performances in the programs (where one particular program gives high rank percentages, but not other programs) can be avoided. Using First Quartile and Third Quartile also help us to gain more insight on the spread of the bug localization performance of the respective programs. The *top-rank-bug score* is a simple measure that examines the proportion of programs having the bug ranked top. *Relative score* is a useful measure to determine the best bug localization performance a metric can achieve.

We have evaluated these performance measures on the single bug programs of the Siemens Test Suite benchmark to gain insights on these measures. For a fair comparison with previous studies, we evaluate our proposed bug localization approaches using rank percentages and report the bug localization performance using the average rank percentages throughout the thesis. We also discussed issues related to empirical evaluation and performance measures, for example, performance measures may be unreliable if datasets are not chosen properly.

5

A Model for Spectra-Based Software Fault Diagnosis

5.1 Introduction

We have discussed in earlier chapters, various methods based on spectra metrics for bug localization. Unfortunately, we cannot determine whether there exist an optimal metric that outperforms all the proposed metrics. In this chapter, we present a simple model program to capture the insights of software fault diagnosis. The motivation of this approach is to understand the behaviour of single bug programs and develop an optimal spectra metric. A large number of metrics have been proposed and we observe some of these metrics give identical performance results to some other metrics. In order to study the equivalence of these metrics, we develop the notion of equivalence for two different metrics that produce similar ranking. Using this principle, we detail the equivalence of the spectra metrics, which are listed in Table 2.3. We describe our model program and detail our methodology for considering multisets of execution paths of the model program. We also describe how we evaluate the performance of spectra metrics in this model program. We discuss optimal ranking and give some insights into several of the spectra metrics used in our evaluation. This is followed by the evaluation of all the spectra metrics using the proposed model program and the empirical benchmarks (the Siemens Test Suite, subset of the Unix Test Suite, Concordance and Space programs). We also evaluate multiple-bug programs from the benchmarks and detail some discussions of using model program for multiple-bug programs. Finally, we evaluate single bug programs of the Siemens Test Suite using other performance measures, which we have detailed in Chapter 4.

5.2 Spectra Metrics and Their Equivalences

Spectra-based ranking has been applied to software fault diagnosis by various researchers. We have presented a comprehensive list of spectra metrics in Table 2.3. The predicate-based spectra metrics [Liblit et al., 2005] can be found in Table 2.5. In this thesis, we adapt the predicate-based metrics introduced by Liblit et al. in the CBI system [Liblit et al., 2005] to statement-based metrics. Instrumented statement execution provides the same information as the instrumented predicates if they are related to a control flow. For example, in the program segment *S1; if B then S2*, we know B is *observed* (reach) if and only if *S1* is executed, and B is *True* if and only if $S2$ is executed. Note that although the information about B is available, it is spread across two different statements. The program spectra methods we consider, which never combine the a_{ij} values from the different statements may not always be able to reproduce the predicate-based ranking.

The $ITE2_8$ model program we introduce in Section 5.3 is very simple: For a given set of tests, the $Context$ (one of the predicate-based spectra metrics in Table 2.5) is the same for all control flow predicates; they are always *observed*. This allows the predicate-based spectra metrics used in the CBI system to be translated into the statement-based spectra formalism and compared fairly with other metrics; there is a bijection between statements and predicates and our translation of the formulas preserves the ranking in all cases. The definitions of these metrics are given in Table 2.3. We use the same formulas in our empirical experiments. However, for the test suite programs, $Context$ can vary between different predicates. This means our translation of the CBI predicate-based metric to statement-based metric does not preserve ranking, so the empirical performance we report for these metrics (Section 5.8) may not reflect that of the CBI system. Recently, study of ours addressed this concern by proposing several heuristics to reconstruct predicate-based spectra coverage from statement-based spectra coverage [Naish et al., 2010].

5.2.1 Equivalence of the Spectra Metrics

When we evaluate spectra metrics on our benchmarks in Section 5.8, we notice several spectra metrics produce the same ranking, which results in same bug localization performance. In this section, we prove the equivalence of these metrics. Dice is simply twice Jaccard, for example. Pearson and Phi metrics share the same formula. Jaccard, Levandowsky, and Watson metrics are also equivalent for ranking, as they share an identical formula. For others, it is more subtle. For the sake of simplifying the proofs, a simpler terminology is used for some of the terms that have been defined in the Glossary. F and P denote the number of fail and pass tests, in place of $totF$ and $totP$ respectively. T

denotes the total number of tests; $T = F + P$. $a_{nf} = F - a_{ef}$ and $a_{np} = P - a_{ep}$ are denoted in some of the propositions below.

Lemma 5.2.1. *A spectra metric $m(\bar{a})$ produces the same rankings as $f(m(\bar{a}))$ if f is a monotonically increasing function.*

Proof. For any given two statements, S_1 and S_2, S_1 ranks higher than S_2 if and only if $m(S_1) > m(S_2)$. Since f is a monotonically increasing function, $f(m(S_1)) > f(m(S_2))$. Therefore, f will rank S_1 higher than S_2. □

Based on the above lemma, we establish several spectra metrics that are equivalent for ranking by the following propositions. In the propositions, the monotonically increasing functions are defined within the range of the metric values evaluated on respective spectra metrics. For example, statements evaluated using Jaccard metric can only range from 0 to 1. Therefore, a monotonically increasing function defined for Jaccard metric is only true within the range of 0 to 1.

Proposition 5.2.2. *The Jaccard, Anderberg, Sneath & Sokal 2, Sørensen-Dice, Dice, Goodman, Levandowsky, and Kulczynski1 metrics are all equivalent for ranking.*

Proof. We show they are all equivalent to Kulczynski1, $a_{ef}/(a_{nf} + a_{ep})$. For the Jaccard metric, we can apply the monotonically increasing function $f(x) = \frac{1}{(1/x)-1}$ to obtain the same result. We can apply the monotonically increasing function $\frac{2}{(1/x)-1}$ for the Anderberg metric. Sneath & Sokal 2 and Anderberg share an identical formula; hence, equivalent. For the Sørensen-Dice metric, we can apply monotonically increasing function $\frac{1}{(2/x)-2}$. For Dice, we can apply monotonically increasing function $\frac{1}{(2/x)-1}$. For Goodman, we can apply monotonically increasing function $\frac{1}{(1/x)-1} + \frac{1}{2}$. Levandowsky metric has the same formula as Jaccard. □

Proposition 5.2.3. *The Rogers and Tanimoto, Gower1, Gower2, Simple Matching, SokalDist, Hamann, Sneath & Sokal 1, M1, M3, Wong2, Euclid, Hamming, NFD, Manhattan, and Lee metrics are all equivalent for ranking.*

Proof. Simple Matching is equivalent to $(a_{ef} + a_{np})/T$. Applying the monotonically increasing function $f(x) = x \cdot T - P$, we obtain $a_{ef} - a_{ep}$ (Wong2). Hamann is equivalent to $(2a_{ef} + 2a_{np} - T)/T$ and applying the monotonically increasing function $(x \cdot T + T)/2T$ we get the same result. For Rogers and Tanimoto, we can apply monotonically increasing function $\frac{2}{(1/x)+1}$ to get the Simple Matching formula. For Sneath & Sokal 1, monotonically increasing function $\frac{1}{(2/x)-1}$ can be applied to get the same result. For M1, we can apply monotonically increasing function $\frac{1}{(1/x)+1}$ to obtain the Simple Matching formula. Euclid squared is simply the same as Hamming, NFD, Manhattan, and Lee. Applying

monotonically increasing function x/T to Hamming, we obtain Simple Matching. We can transform Simple Matching by $2x$ to obtain the M3 formula. We square SokalDist $\sqrt{\frac{a_{ep}+a_{nf}}{T}}$ to obtain the Simple Matching formula. By using the notation of $a_{nf} = F - a_{ef}$, $a_{np} = P - a_{ep}$ and simplifying the Gower1 formula, we obtain $\frac{2(a_{ef}+a_{np})-T}{T}$. Adding a constant value of 1 to the latter formula will obtain $\frac{2(a_{ef}+a_{np})}{T}$. Dividing by two on the latter equation gives Simple Matching. Gower2 is equivalent to the Simple Matching using the monotonically increasing function $f(x) = \frac{0.5}{\frac{1}{x}-0.5}$. □

Proposition 5.2.4. *The Scott and Rogot1 metrics are equivalent for ranking.*

Proof. Taking the Scott formula, $\frac{4(a_{ef}a_{np}-a_{nf}a_{ep})-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$ and using a_{nf} and a_{np} notations, by simplifying the latter formula, we obtain

$$\frac{-F^2 + (4P + 2F)a_{ef} - 2Fa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}{F^2 + 2PF + 2Pa_{ef} + 2Pa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}$$

Adding one to the formula, we obtain

$$\frac{2PF + (6P + 2F)a_{ef} + (2P - 2F)a_{ep} - 4a_{ef}a_{ep} - 2a_{ep}^2 - 2a_{ef}^2}{F^2 + 2PF + 2Pa_{ef} + 2Pa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}$$

Taking the Rogot1 formula, $\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}} + \frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$ and simplifying the formula, we obtain the former result divided by two. □

Proposition 5.2.5. *The Ochiai, Ochiai3, and CorRatio metrics are equivalent for ranking.*

Proof. The Ochiai metric equals $\frac{a_{ef}}{\sqrt{totF(a_{ef}+a_{ep})}}$; by squaring the metric, we obtain Ochiai3. The CorRatio metric shares an identical formula to Ochiai3; hence equivalent. □

Proposition 5.2.6. *The Tarantula, CBI Increase (CBI Inc), q_e, and Coef metrics are equivalent to $\frac{a_{ef}}{a_{ep}}$ for ranking.*

Proof. The Tarantula metric is equivalent to $\frac{a_{ef}/F}{a_{ef}/F+a_{ep}/P}$. Applying monotonically increasing function $\frac{F}{P((1/x)-1)}$, we obtain $\frac{a_{ef}}{a_{ep}}$. CBI Increase is equal to $\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{F}{T}$. We apply monotonically increasing function $1/(1/(x + \frac{F}{T}) - 1)$ to obtain $\frac{a_{ef}}{a_{ep}}$. q_e is equivalent to $\frac{a_{ef}}{a_{ef}+a_{ep}}$ [Abreu et al., 2007]. Applying monotonically increasing function $\frac{x}{1-x}$, we obtain $\frac{a_{ef}}{a_{ep}}$. The Coef metric shares an identical formula to q_e; hence, equivalent. □

The proposition for Tarantula and CBI Increase metrics is of particular interest for two reasons. First, it illustrates another advantage of our more formal approach. Our initial implementation of the CBI Increase (CBI Inc) metric had slightly different performance to Tarantula. Having proved that they should be identical, the code was examined and a bug

in our implementation of the CBI Increase (CBI Inc) was found[1]. Second, the Tarantula metric does not perform particularly well — there are significantly better metrics. This suggests that there is a strong possibility of improving the performance of the CBI system [Liblit et al., 2005] by using a better metric. Alternatively, the role of $Context$ may be particularly important and this could influence the way data on statement executions are best used for diagnosis. In our recent study, we found that $Context$ played an important role in improving the performance of the bug localization, using a predicate-based metric known as FPC ($Failure$ plus $Context$), as compared to using the CBI Increase (CBI Inc) [Naish et al., 2010].

Both Tarantula and q_e metrics have been empirically evaluated on the benchmark of the Siemens Test Suite and the subset of the Unix Test Suite [Lee et al., 2009a] and found to produce the same ranking. Error detection accuracy, q_e, has been proposed by Abreu et al. to determine the accuracy of the bug being detected, given that the bug is known [Abreu et al., 2007]. q_e is later known as bug consistency; details are deferred to Chapter 6.

Proposition 5.2.7. *The Braun, Interest, and Forbes metrics are equivalent for ranking.*

Proof. The Interest metric equals $\frac{a_{ef}}{(a_{ef}+a_{ep})F}$. Simplifying the latter formula and multiplying the formula with F, we obtain the Braun metric. The Forbes metric equals $\frac{Ta_{ef}}{(a_{ef}+a_{ep})F}$. Multiplying the metric with the constant of $\frac{1}{T}$ is equivalent to the Interest metric. □

Proposition 5.2.8. *The Russell and Rao, Simpson, and Wong1 metrics are equivalent for ranking, and, if there is a single bug, they result in the same ranking for the bug as the Binary metric.*

Proof. The Russell and Rao metric equals a_{ef}/T; multiplying the latter formula by T gives a_{ef} (which is Wong1). This is maximised for exactly those statements that are executed in all fail tests. There is at least one such statement, the buggy one, if there is a single bug and at least one fail test; if there are no fail tests all statements are ranked equally for all these metrics. The statements that are executed in all fail tests have $a_{nf} = 0$, and thus also maximise the Binary metric. So the same set of statements is maximally ranked in these metrics and includes the bug, if there is a single bug. The Simpson metric equals a_{ef}/F; multiplying by F gives a_{ef} (which is Wong1). □

Note the non-maximal rankings using Binary can differ from those using Russell and Rao, Simpson, and Wong1. For multiple-bug programs, the performance may differ. Our empirical results in Section 5.8 shows identical results for these metrics. O and O^p have a similar relationship, but can give multiple distinct ranks for statements with $a_{nf} = 0$:

[1] A special case to avoid division by zero returned a value that was not minimal in all cases.

Proposition 5.2.9. *O and O^p give the same rankings to all statements with $a_{nf} = 0$, which includes the bug if there is a single bug.*

Proof. If $a_{nf} = 0$, O is a_{np} and O^p is $a_{ef} - a_{ep}/(P+1)$. Also, both metrics return higher values than for any statement with $a_{nf} > 0$. Applying monotonically increasing function $aef - (P-x)/(P+1)$ to O gives O^p. □

5.3 Model Program

We introduce a model program in order to gain insights into the different execution paths of a typical program. We can apply the spectra metrics (see Table 2.3) on the model program to compare bug localization performance of these metrics. The model program also enables us to understand and develop optimal metrics for single bug programs.

We introduce a simple model program in Figure 5.1. The figure gives the code for the If-Then-Else-2 (*ITE2*) program segment, which we use as a model for single-bug programs. It is assumed that functions `s1()` and `s2()` exist and may have side effects such as assignments to variables whereas Boolean functions `t1()`, `t2()`, and `t3()` return Booleans but have no side effects. The intention is that *ITE2* should assign `True` to variable `x`. There are also intentions for the individual statements and these are met by the program except for *S4*, which may sometimes assign `False` instead of `True` to `x`.

```
if (t1())
    s1();          /* S1 */
else
    s2();          /* S2 */
if (t2())
    x = True;      /* S3 */
else
    x = t3();      /* S4 - BUG */
```

Figure 5.1: Program Segment If-Then-Else-2 (ITE2)

We use `t3()` to model the fact that the buggy code may sometimes behave correctly and sometimes trigger a failure. `t2()` (and the second if-then-else) is used to model the fact that the buggy code may not be executed in every run. We use `t1()` to model the fact that spectra metrics (and debugging in general) must cope with *noise*. It may be that the execution of *S1*, for example, is strongly correlated with fail test runs. This may be due to logical dependencies within the program or the particular selection of test data. The *signal* we want to detect is associated with `t2()` — the buggy statement is executed if and only if `t2()` returns `False`. The signal is essentially attenuated by `t3()` — if

`t3()` almost always returns `True`, there is little signal we can detect (and its more likely that the noise will be greater, leading to *S1* or *S2* being ranked top).

Our intuition suggests that having noise and an attenuated signal are the two most important features we need in a model, and *ITE2* is the simplest model program we can think of that has these features. Despite its simplicity, this model has been very useful in evaluating, understanding, and improving the performance of spectral diagnosis methods, as our later results show. There are many ways the model could be extended, for example, by:

- having more bugs,
- having more sources of noise,
- simulating loops, so both branches of an if-then-else can be executed in a single test, and
- having statements that are executed more or less often over typical sets of tests (in $ITE2$, all statements are executed in half the tests, on average).

The way we evaluate performance of metrics (described in the next section) is independent of the model. We have experimented with all these extensions and report some general observations here.

5.4 Performance Evaluation using Multisets of Execution Paths

We use the term *execution path* to refer to the set of statements executed for a particular test, along with the result of the test. A single test case determines the execution path (for deterministic programs at least). A set of test cases determines a *multiset* of execution paths (two or more distinct test cases might result in the same execution path), which determines the a_{ij} values for each statement and the performance of a given metric for that set of tests cases. We abstract away the details of test cases (and sidestep the issue of nondeterminism) and focus on multisets of execution paths. A typical metric ranks the buggy statement highly for some multisets of execution paths but not for others. Ideally, we would like a test set to give an even coverage of all execution paths but in reality, test sets are often far from this ideal. Our methodology for evaluating the overall performance of a metric uses the number of tests as a parameter. Given a number of tests T, we determine the average performance over *all possible multisets* of T execution paths. Table 5.1 and Table 5.2 in pages 111–112 give results from our model for each metric for various

```
if (t3()) x = True; else x = False;     /* S4 - BUG */
```

Figure 5.2: Coding of *S4* with Two Explicit Paths

values of T. We choose between 2 and 1000. When the number of possible multisets is not too large, we generate each distinct multiset once to evaluate the overall performance of each metric. For larger numbers of multisets, we compute estimates by randomly sampling multisets from a uniform distribution.

5.4.1 Generating Multisets of Execution Paths

There is a naive way of generating a random multiset of execution paths — simply execute the model program the required number of times with each predicate being a random choice of true or false. However, this leads to extremely skewed distributions. For example, consider a program with just two paths; one correct and one that always results in failure. With 500 tests, there are 501 possible multisets of execution paths (with zero up to 500 fail tests). However, the probability of generating the one with 500 fail tests is $1/2^{500}$, rather than $1/501$. It is hard to determine the most realistic distribution, but a uniform distribution is the simplest assumption we can make conceptually (though not the simplest to implement). Other distributions can easily be generated from a uniform distribution, or sub-sets of the total space can be examined (which is what we report in our experiments). Even without refining the distribution, the results obtained from our model fit reasonably well with empirical results.

To distinguish between fail and pass tests, the two possible return values of `t3()` in $ITE2$ are treated as two separate execution paths. This is equivalent to re-coding statement *S4* as in Figure 5.2. For the first if-then-else of $ITE2$, there are two possible paths: *S1* and *S2*. For the second if-then-else, there are three possible paths: *S3* (always returning `True`), *S4* returning `True` and *S4* returning `False`. This gives a total of six execution paths of interest for the whole program segment. However, *S4* is used in twice as many paths as *S3*. To remove this bias, we also treat *S3* as having two paths (both correct), giving a total of four possible paths for the second if-then-else and eight paths overall. We use this model in our experiments and refer to the model as $ITE2_8$. Increasing the number of execution paths can greatly increase the number of multisets. Since we use random sampling, though it has relatively little effect on our ability to evaluate metrics.

The system we use to conduct our experiments supports a high level definition of the model program code including the number and placement of buggy statements, program points that lead to test case failure and duplication of execution paths. This has allowed us to experiment with a wide range of other models. A small Prolog program converts

```
#define MODEL_NAME "ite2_8"
#define MODEL_CODE "\
        if 1/2=1+1 then \
            S1\
        else \
            S2\
        if 2/3=2+2 then \
            S3\
        else \
            S4\
            if 1/2=1+1 then \
                fail\
            else "
#define NBUGS 1
#define NSTATS 4
#define NPATHS 8
#define NFAILPATHS 2
#define INIT_P_RESULT {1, 1, 0, 0, 0, 0, 0, 0}
#define INIT_P_USED {1,0,1,0,1,0,0,1,1,0,1,0,1,0,0,1,\
0,1,1,0,0,1,1,0,0,1,0,1,0,1,0,1}
#define PATH_COUNTS \
totfails  = 0+nt[0]+nt[1]; \
totpasses = ntests - totfails; \
fails[1]  =  0+nt[0]; \
passes[1] = 0+nt[2]+nt[4]+nt[5]; \
fails[2]  =  0+nt[1]; \
passes[2] = 0+nt[3]+nt[6]+nt[7]; \
fails[3]  =  0; \
passes[3] = 0+nt[4]+nt[5]+nt[6]+nt[7]; \
fails[4]  =  0+nt[0]+nt[1]; \
passes[4] = 0+nt[2]+nt[3]; \
```

Figure 5.3: C Macros for the $ITE2_8$ Model Program

the model definition to C macros. The macros are included in a C program that generates multisets and computes the performance of different metrics.

A Prolog program is used to produce C macros for the $ITE2_8$ model program shown in Figure 5.3. The code defines the data structure of the model program with the number of paths taken for each statement. These statements are represented with statement 1 as S1, statement 2 as S2, statement 3 as S3, and statement 4 as S4 (which is the bug). These macros also define possible paths in each if-then-else, for example, the first if-then-else is defined by $1/2 = 1 + 1$. $1/2$ refers to half of the time the $S1$ and $S2$ is executed respectively. $1 + 1$ refers to one possible path taken each for $S1$ and $S2$. The

code also defines the number of bug(s) in the model program, number of statements, the total number of execution paths, and the statement that is buggy. The total number of fail paths for the model program is defined as $NFAILPATHS$. $INIT\_P\_RESULT$ represents the paths of the model program that pass (0) or fail (1). $INIT\_P\_USED$ defines the paths executing each statement in binary form. 1 indicates the statement has been executed by a particular path. 0 indicates a particular path does not execute the statement. $PATH\_COUNTS$ consists of several assignment statements for the total pass and fail test cases and the passes and fails for the respective statements of the program. Model programs can be defined with different data structures (such as nested if-then else), number of paths that pass or fail, number of statements, and number of bugs. Based on this information, model programs can be generated automatically using the Prolog program.

Given p ($\geq$ 1) execution paths in the program (8 for $ITE2_8$) and T ($\geq$ 0) tests, the number of distinct multisets of execution paths, $f(T, p)$, can be determined as follows. A multiset can be represented as a sequence of natural numbers of length p (the execution count for each path) that sum to T. If $T = 0$ or $p = 1$, only one multiset is possible. In general, the first number can be anything from 0 to T, and the rest of the sequence is length $p - 1$, so we can use the recurrence:

$$f(T, p) = \sum_{i=0}^{T} f(i, p - 1)$$

The problem is equivalent to the more classical "balls and pins" or "books and bookends" problems (the number of ways T books can be partitioned on a shelf using $p - 1$ bookends) and there are several equivalent recurrences. The solution is the Binomial numbers (also known as Pascal's triangle) — $f(T, p) = C(T + p - 1, T)$ where

$$C(n, k) = \frac{n!}{k!(n - k)!}$$

We use a bijection between integers in the range 1 to $f(T, p)$ and sequences of p integers that sum to T. We repeatedly generate a random number in this range (or generate each number once if the range is small enough) and map it to a sequence of path counts which represents a multiset.

5.4.2 Using Top-rank-bug Score to Evaluate Performance

For each sequence (multiset), the a_{ij} are determined. Suppose M is a spectra metric, X is a multiset of execution paths, and the spectra metric is evaluated for each statement. m_1, m_2, m_3, and m_4 are M applied to the a_{ij} tuples corresponding to X for statements $S1$, $S2$, $S3$, and $S4$, respectively. We determine the *top-rank-bug score* for that multiset (we use

this scoring function for $ITE2_8$; the details of this function can be found in Subsection 4.4.2 of Chapter 4).

For example, suppose $T = 5$ and there is one execution of *S1*;*S4* that fails, two executions of *S2*;*S4* that pass, and two executions of *S1*;*S3* that pass. For *S1* and *S4*, $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$ would be $\langle 2, 0, 2, 1 \rangle$ and for *S2* and *S3* it would be $\langle 2, 1, 2, 0 \rangle$. Using the Jaccard metric, m_1, m_2, m_3, and m_4 would be $1/3$, 0, 0, and $1/3$, respectively. $S4$ (the bug) and $S1$ would be ranked equal-highest. Using the *top-rank-bug score* (Subsection 4.4.2), the score would be $1/2$.

Definition 14 (Total score for spectra metric M with T tests)**.** *The* total score *for spectra metric M with test set size T is the sum of the top-rank-bug scores for M with paths X, over all possible multisets X of T execution paths that contain at least one fail path.*

In our experiments, we report percentage scores: the total *top-rank-bug score* for all multisets examined divided by the number of multisets, times 100. We give percentage scores for multisets of paths with certain characteristics (such as particular numbers of fail cases). Multisets of paths without these characteristics are simply ignored when computing percentage scores. We always ignore multisets with no fail tests. For example, with a single test, there are just eight multisets of paths. Six of these have no fail tests and are ignored. The remaining two multisets have a single execution path containing $S1$ or $S2$ followed by (the failing path through) $S4$. For both of these multisets, the Jaccard metric (and other reasonable metrics) gives a score of 0.5. The percentage score is thus $(0.5 + 0.5)/2 \times 100 = 50\%$.

5.5 Optimal Ranking

For any given multiset of paths, metrics exist that rank $S4$ top (or equal top) but no single metric exists that ranks *S4* top for all multisets. Thus no metric is best in all cases. However, it is possible to have metrics that are optimal in the sense that they maximise the total score over all possible multisets.

Definition 15 (Optimal spectra metric for T tests)**.** *A spectra metric is* optimal *for test set size T if no other spectra metric has a higher total score with T tests.*

This definition is most appropriate if we assume a uniform distribution, since each multiset of paths is considered equally. It would be possible to generalise the definition so each multiset has a *weight*, which is multiplied by the score for that multiset. Ignoring some multisets, as we do in our experiments, is equivalent to having a weight of zero for those multisets and a weight of one for other multisets. Note that a metric that is optimal

for a uniform distribution (equal weights) may not be optimal for other distributions, but may be close to optimal.

5.5.1 Optimal Spectra Metric O

Definition 16 shows our proposed spectra metric O, and we prove that it is optimal (for $ITE2_8$) for all numbers of tests.

Definition 16 (Spectra metric O)**.**

$$O(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} -1 & \text{if } a_{nf} > 0 \\ a_{np} & \text{otherwise} \end{cases}$$

Superficially this is a rather odd metric as it only uses a_{nf} and a_{np}, which appear to be the least important variables in most of the other metrics. However, for single bug programs such as $ITE2_8$, a_{nf} is always zero for the buggy statement. Any statement with a non-zero a_{nf} can be given the lowest rank. Furthermore, any two statements that have $a_{nf} = 0$ must have the same a_{ef} value since $a_{nf} + a_{ef}$ is the total number of fail tests (which is the same for all statements). Similarly, $a_{np} + a_{ep}$ is the total number of pass tests, so there is only one non-trivial degree of freedom. There is no *a priori* reason to suppose a buggy statement is executed more or less often than a correct statement. However, since all fail tests use the buggy statement, a_{ep} will tend to be smaller for the buggy statement, so a_{np} will tend to be higher.

Proposition 5.5.1. *For single bug programs, increasing the value returned by a spectra metric when $a_{nf} > 0$ never increases the total score for the metric.*

Proof. Since there is a single bug, a_{nf} is greater than 0 only for non-buggy statements. Increasing the rank of a non-buggy statement never increases the score for a multiset, so the total score is never increased. □

The key lemma below shows that O cannot be improved by making a minimal change to the ranking it produces. O ranks a statement with $a_{nf} = 0$ and $a_{np} = x$ lower than a statement with $a_{nf} = 0$ and $a_{np} = x + 1$, and no statements are ranked between these. We consider the effect of swapping the ranks of two such statements, or making the ranks equal.

Lemma 5.5.2. *Suppose O' and O'' are spectra metrics such that for a single x*

$$O'(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} x + 1 & \text{if } a_{np} = x \text{ and } a_{nf} = 0 \\ O(a_{np}, a_{nf}, a_{ep}, a_{ef}) & \text{otherwise} \end{cases}$$

$$O''(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} x + 1.5 & \text{if } a_{np} = x \text{ and } a_{nf} = 0 \\ O(a_{np}, a_{nf}, a_{ep}, a_{ef}) & \text{otherwise} \end{cases}$$

For $ITE2_8$, O has a total score at least as high as O' and O'' for all test set sizes T.

Proof. For O'' the ranking of statements where $a_{np} = x$ and $a_{np} = x + 1$ is swapped compared to the ranking using O, and for O' they have equal rank; the relative rank of all other pairs of statements is the same as that using O. Since $S3$ is not executed in the fail test(s) it has a metric value of -1, which is strictly less than the value for $S4$ and will not affect the score for any multiset. Similarly, at least one of $S1$ and $S2$ have a score of -1 for each multiset: whenever $S1$ is used $S2$ is not used and vice versa, so they can't both be executed in (all) the fail test(s). We can never have a tie between $S1$ and $S2$ unless both have the value of -1.

Thus O has a total score greater or equal to that of O' and O'' if and only if the number of multisets such that $\langle a_{np}, a_{nf} \rangle = \langle x + 1, 0 \rangle$ for $S4$ and $\langle a_{np}, a_{nf} \rangle = \langle x, 0 \rangle$ for $S1$ or $S2$ (whichever has the larger score) is greater or equal to the number of multisets such that $\langle a_{np}, a_{nf} \rangle = \langle x + 1, 0 \rangle$ for $S1$ or $S2$ (whichever has the larger score) and $\langle a_{np}, a_{nf} \rangle = \langle x, 0 \rangle$ for $S4$. Due to the symmetry between $S1$ and $S2$, we can simply take the numbers of multisets where $S1$ has the larger score, and double it. All these multisets have $a_{nf} = 0$ for $S1$ and $S4$, so we leave this constraint implicit below.

The six pass paths in the $ITE2_8$ can be classified as follows: two include $S1$ but not $S4$, one include $S4$ but not $S1$, two include neither $S4$ or $S1$ and one include $S4$ and $S1$ (due to symmetry the same holds if we replace $S1$ by $S2$ here). Let b be the number of tests using neither $S4$ or $S1$ (it contributes to the a_{np} total for *both* since a_{np} is the number of pass tests that *do not* execute the statement). Also, let $y = x - b$ and $z = T - 2y - b - 1$.

The number of multisets such that $a_{np} = x + 1$ for $S4$ and $a_{np} = x$ for $S1$ is the number of multisets where there are $y + 1$ paths that contribute only to a_{np} for $S4$, y paths that contribute only to a_{np} for $S1$, b paths that contribute to both, and the remaining z paths that contribute to neither. The number of distinct paths in these categories in $ITE2_8$ are two, one, two, and one, respectively (as noted above), and b can range between zero and $\lfloor T/2 \rfloor$ so the number of multisets is

$$\sum_{b=0}^{\lfloor T/2 \rfloor} f(y+1, 2) f(y, 1) f(b, 2) f(z, 1)$$

Similarly for $S2$. By the same reasoning, the number of multisets such that $a_{np} = x+1$ for $S1$ and $a_{np} = x$ for $S4$ is

$$\sum_{b=0}^{\lfloor T/2 \rfloor} f(y, 2) f(y+1, 1) f(b, 2) f(z, 1)$$

Similarly for $S2$. It is sufficient to show that $f(y+1,2)f(y,1) \geq f(y,2)f(y+1,1)$ in all cases. Using definitions of f and C and simplifying we obtain

$$C(y+2,y+1)C(y,y) \geq C(y+1,y)C(y+1,y+1)$$

$$C(y+2,y+1) \geq C(y+1,y)$$

$$y+2 \geq y+1$$

which is true. □

Theorem 5.5.3 (Optimality of O)**.** *Spectra metric O is optimal with respect to $ITE2_8$ for all test set sizes.*

Proof. Suppose some spectra metric O' differs from O. Due to Proposition 5.5.1, any difference in ranking for tuples such that $a_{nf} > 0$ cannot make O' better than O. For tuples in which $a_{nf} = 0$, a_{ef} is the number of fail tests, which is the same for all tuples, and $a_{np} + a_{ep}$ is the number of pass tests. Thus, the number of distinct values for the tuples is the number of distinct values for a_{np}, and O' is equivalent to a function that maps these tuples to the range of a_{np} values. O can be modified to obtain an equivalent ranking in a finite number of steps by swapping the ranking of adjacent values or making them equal. But by Lemma 5.5.2, none of these steps would increase the total score. □

5.5.2 Other Optimal Spectra Metrics

The proofs above can be generalised to show that a spectra metric M is optimal for $ITE2_8$ if, given a fixed number of pass and fail tests,

1. when $a_{nf} = 0$, M is increasing in a_{np} or (equivalently) decreasing in a_{ep} (the key requirement for Lemma 5.5.2), and

2. when $a_{nf} > 0$, the value returned is always less than any value returned when $a_{nf} = 0$ (allowing use of Proposition 5.5.1).

O is the simplest optimal metric from an information-theoretic perspective as it only gives different ranks when necessary. Metrics can also be considered from a geometrical perspective — each metric defines a surface in three dimensions (the four a_{ij} values give just two degrees of freedom if we fix the number of pass and fail tests). Definition 17 shows our proposed O^p metric, which defines a very simple surface — a plane that will be discussed later in Section 5.6. It is optimal for $ITE2_8$ since a_{ef} is maximal when $a_{nf} = 0$ and a_{ep} varies from 0 to at most the number of pass tests, so the fractional component is strictly less than one.

Definition 17 (Spectra metric O^p)**.**

$$O^p(a_{np}, a_{nf}, a_{ep}, a_{ef}) = a_{ef} - \frac{a_{ep}}{P+1}$$

where P is the number of pass test cases.

O^p has the advantage of performing more rationally than O for multiple-bug programs. If there is more than one bug, a_{nf} can be non-zero for *all* statements, leading to O being -1 in all cases. In contrast, O^p ranks statements first on their a_{ef} value and even if this is not maximal, second on their a_{ep} value. O^p and other optimal metrics can be helpful in the comparison of metrics (see Subsection 5.7.5). We also use O^p in our empirical evaluation of metrics (to aid comparison with other studies, some multiple-bug programs are used). In our experiments, we also evaluate the performance of a simplified version of O, called Binary, which ignores a_{np} and allows us to see the relative importance of the two components of O. We introduce a new metric by modifying the denominator of the Jaccard metric. We use the cube root of the existing denominator of the Jaccard metric. Gonzalez has also previously modified the denominator of the Jaccard metric to create the Zoltar metric [A.Gonzalez, 2007]. We also introduce Ample2, which is a variation of the Ample metric that avoids taking the absolute value. A variation of the Wong3 metric, Wong3′, is introduced, where the latter has a special case for statements that are not executed in any test case (motivated by our empirical studies). The definitions of these metrics are shown in Table 2.3.

Although we have formally proved the optimality for $ITE2_8$ only, O and O^p are optimal for a much broader class of single bug programs. Proposition 5.5.1 holds for all single bug programs and the combinatorial argument in the proof of Lemma 5.5.2 can be generalised. With larger numbers of paths and/or sources of "noise", it is typically sufficient to show $\forall y\ f(y+1, j+k)f(y, j) \geq f(y, j+k)f(y+1, j)$, where j and k are positive integers dependent on the number of paths through the program with particular characteristics.

Recently, Debroy et al. propose a grouping approach (details of their study can be found in Chapter 2) which is the same as the O^p metric [Debroy et al., 2010]. In their study, Debroy et al. group statements with identical a_{ef} before sorting the statements within the groups of a_{ef}.

Proposition 5.5.4. *Evaluating statements using the Tarantula metric [Jones and Harrold, 2005] and the grouping approach [Debroy et al., 2010] is equivalent to using O^p metric.*

Proof. In the Tarantula metric, a_{ef} and a_{ep} are part of the metric's numerator and denominator. The grouping approach groups the evaluated statements of the Tarantula metric

(in descending order) based on the a_{ef} [Debroy et al., 2010]. This is essentially giving primary importance to the a_{ef} value. Sorting the statements within the groups of a_{ef} in the grouping approach means giving secondary importance to the -a_{ep} value. Therefore, the grouping approach [Debroy et al., 2010] is equivalent to the O^p metric. Similar principles of importance can potentially be applied for other spectra metrics, such as the Jaccard and Ochiai metrics. □

Xie et al. also propose another grouping approach that helps improve the bug localization performance of single bug programs using several spectra metrics [Xie et al., 2010]. Statements that are not executed by any fail test cases (a_{ef}=0) are assigned a minimum metric value and ultimately ranked on the bottom of the ranking. Xie et al. add the metric value of one to all the statements executed by at least one fail test case. They observe that their grouping approach helps improve bug localization performance using the optimal metric, O^p. The fiddling of the metric values of statements should not affect the performance of bug localization using the O^p metric. This is due to the fact that the O^p metric ranks primarily on a_{ef}, and the buggy statement of single bug programs is executed by all fail test cases. Any changes of the metric value (by adding 1) would not change the ranking of the buggy statement using the O^p metric [2].

5.6 Insights of Spectra Metrics

In the previous section, we have proven that O and O^p are optimal metrics for single bug programs. In this section, we attempt to gain insights and understand several spectra metrics including O, O^p, Zoltar, Wong3, Wong4, Kulczynski2, Tarantula, and Rogers. We show the plot of these metrics' $MetValue$ (z-axis) with respect to a_{ef} (y-axis) and a_{ep} (x-axis). A range of 0 to 100 for a_{ef} and a_{ep} is used.

Figure 5.4 and Figure 5.5 show the surface of the O and O^p metrics. Both surfaces peak on the z-axis ($MetValue$) when the a_{ef} value is high. When the a_{ef} value is at the maximum (100 in this case), the $MetValue$ is maximised. As O metric is defined in two cases (see Definition 16), we observe a flat surface for O metric when the a_{ef} value is not at the maximum. For O^p metric, the surface slants lower as the a_{ef} value is not at the maximum. While the a_{ef} value is at the maximum and the a_{ep} value increases, the surface of O^p metric slants lower ($MetValue$ slightly drops) but it is still higher than any other points on the surface.

[2] We contacted the author for further clarification of their findings. Their experimental results include programs that encountered *runtime errors* (segmentation fault) when `gcov` gathers the coverage. Therefore, the buggy statements in these programs have an a_{ef} of 0. In our study, we obtain the coverage of these programs (without *runtime errors*) from other researchers who use Valgrind [Nethercote and Seward, 2007].

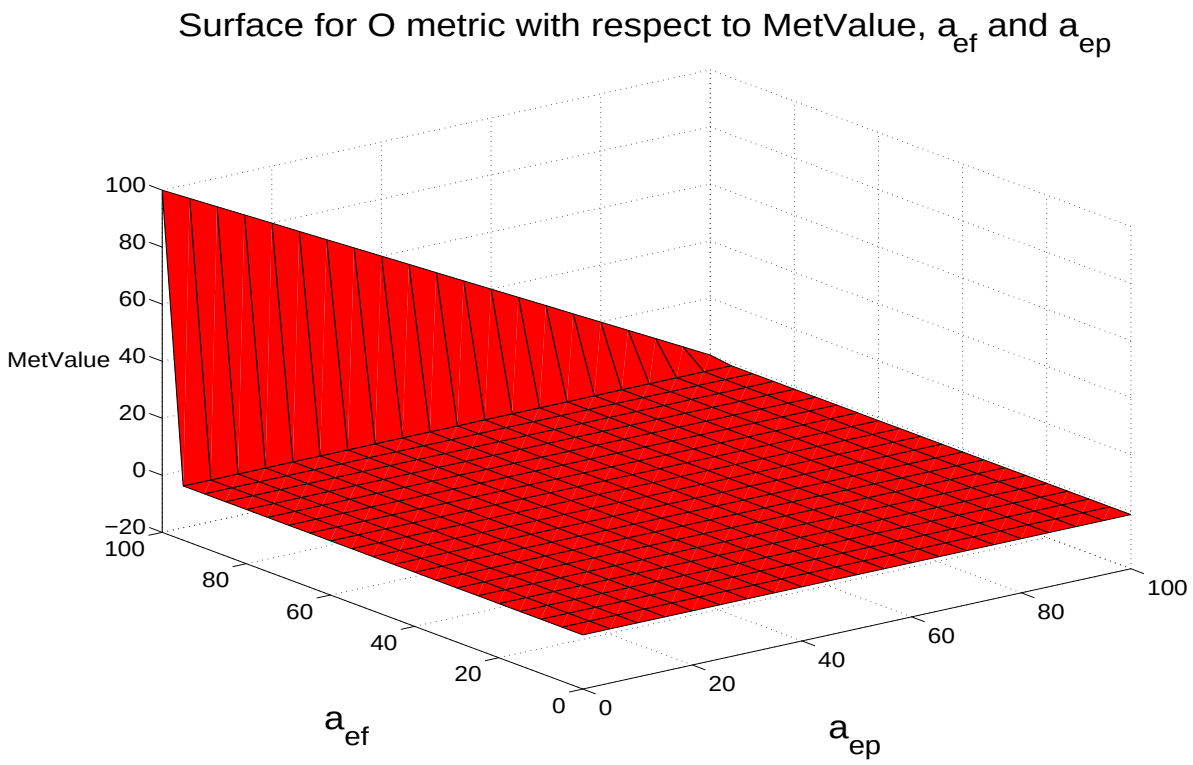

Figure 5.4: Surface for O metric

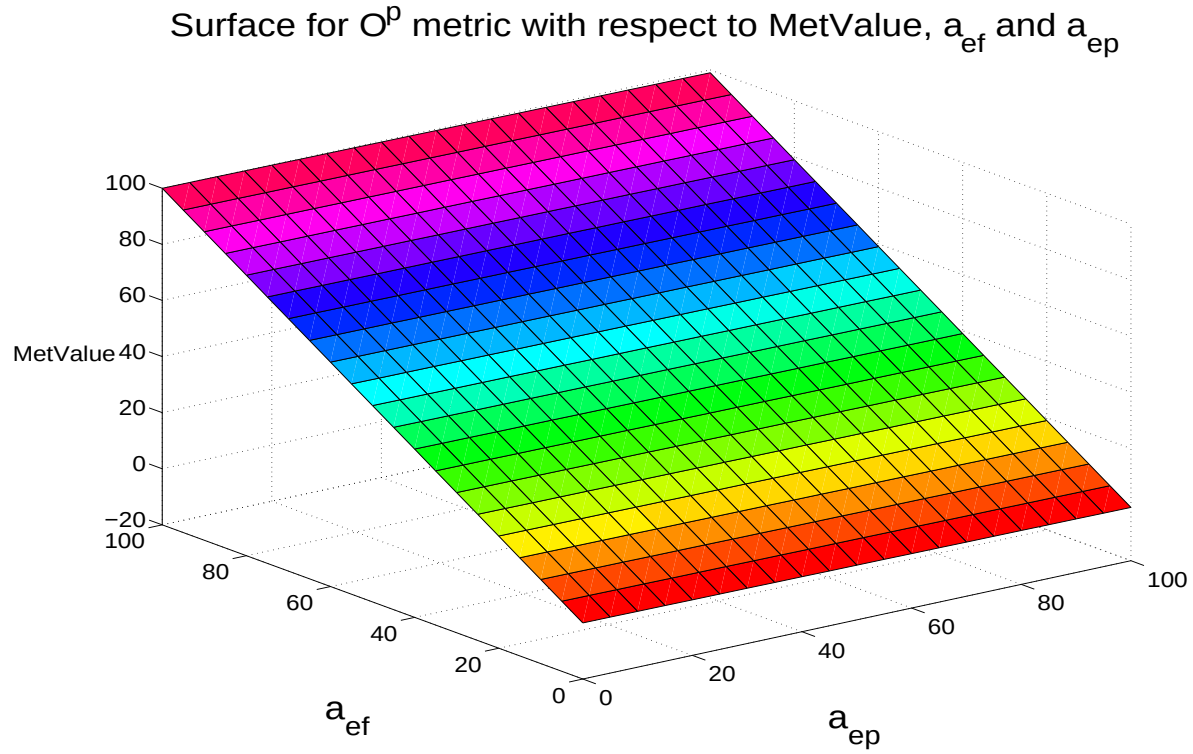

Figure 5.5: Surface for O^p metric

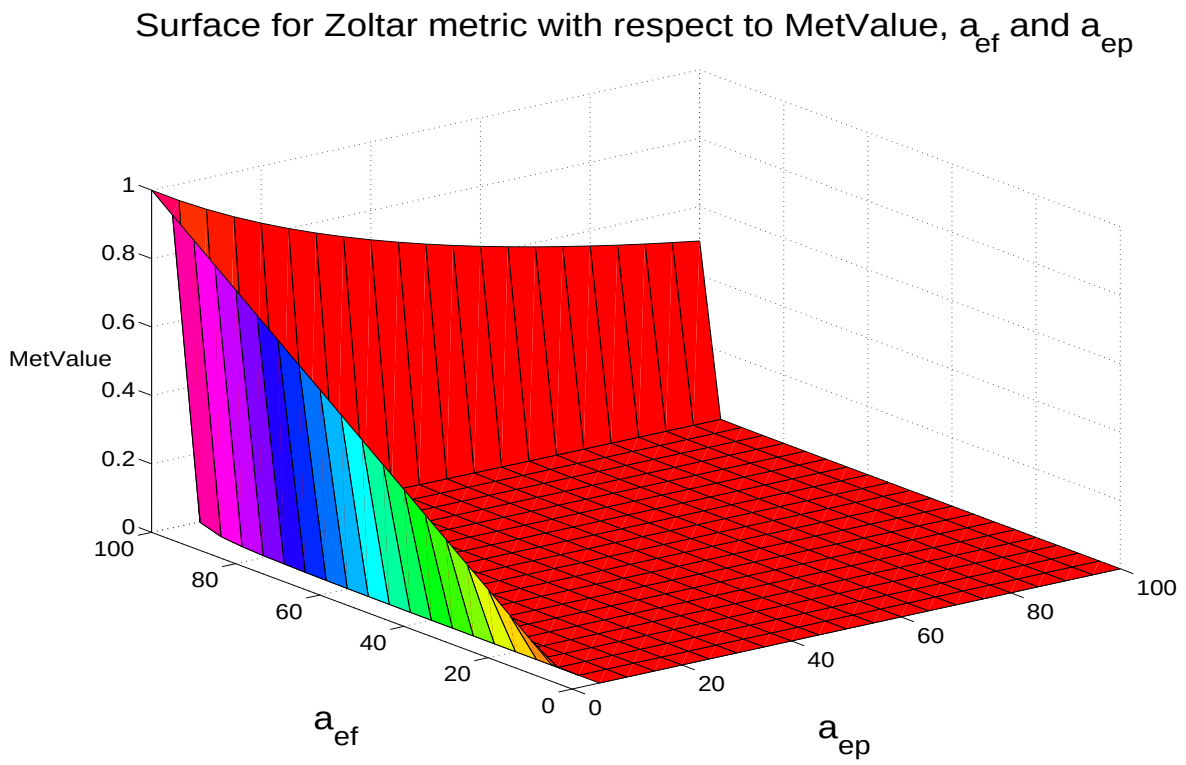

Figure 5.6: Surface for Zoltar metric

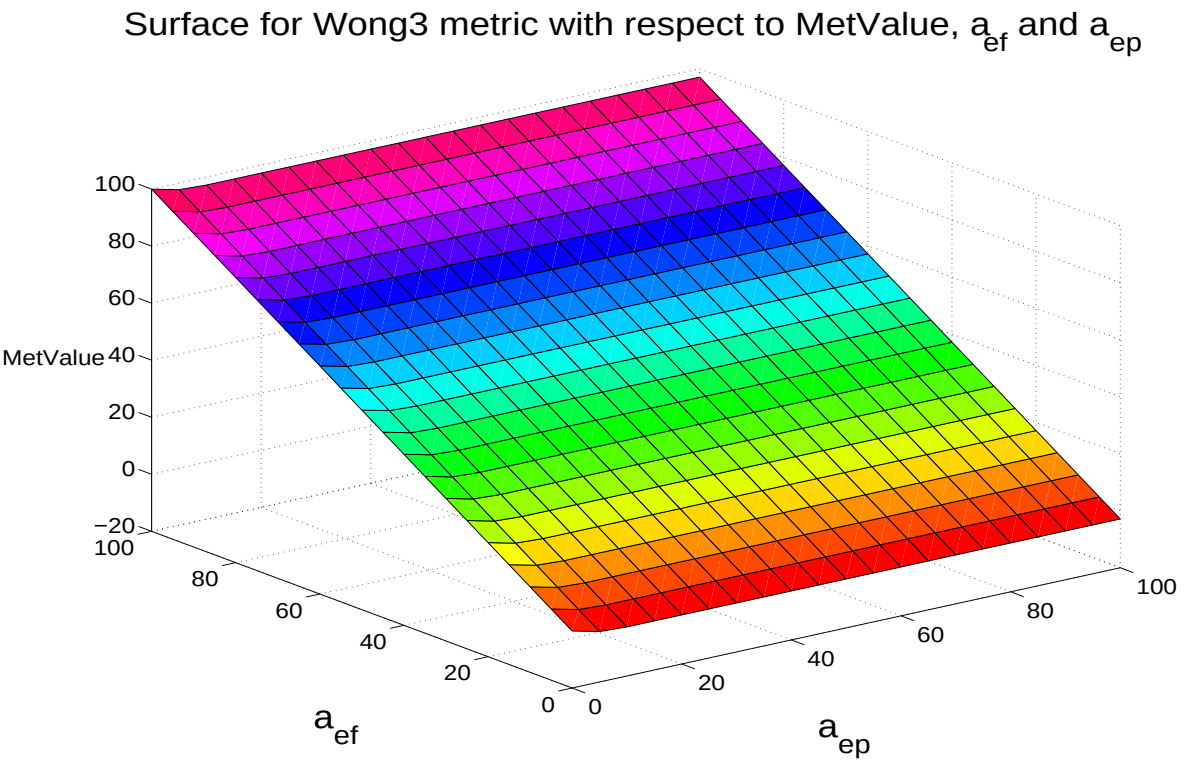

Figure 5.7: Surface for Wong3 metric

If we observe another metric, Zoltar (Figure 5.6), the bottom surface is flat when the a_{ef} is 0. For other cases, the surface elevates from the bottom. The surface peaks on the z-axis ($MetValue$) when the a_{ef} value is high and the a_{ep} value is low.

For the Wong3 metric of Figure 5.7, we observe a surface that is very similar to the O^p metric. However, the surface of this metric lifts slightly higher along the y-axis, a_{ef} when the x-axis, a_{ep} is at 0. The Wong4 metric shown in Figure 5.8 indicates an interesting surface that peaks along the y-axis, a_{ef}, when the x-axis, a_{ep}, is at 0. When the y-axis, a_{ef}, is at 0, the surface skews downward, creating a steep slope. It starts to create an upward slope in the surface when the y-axis, a_{ef}, is at 5.

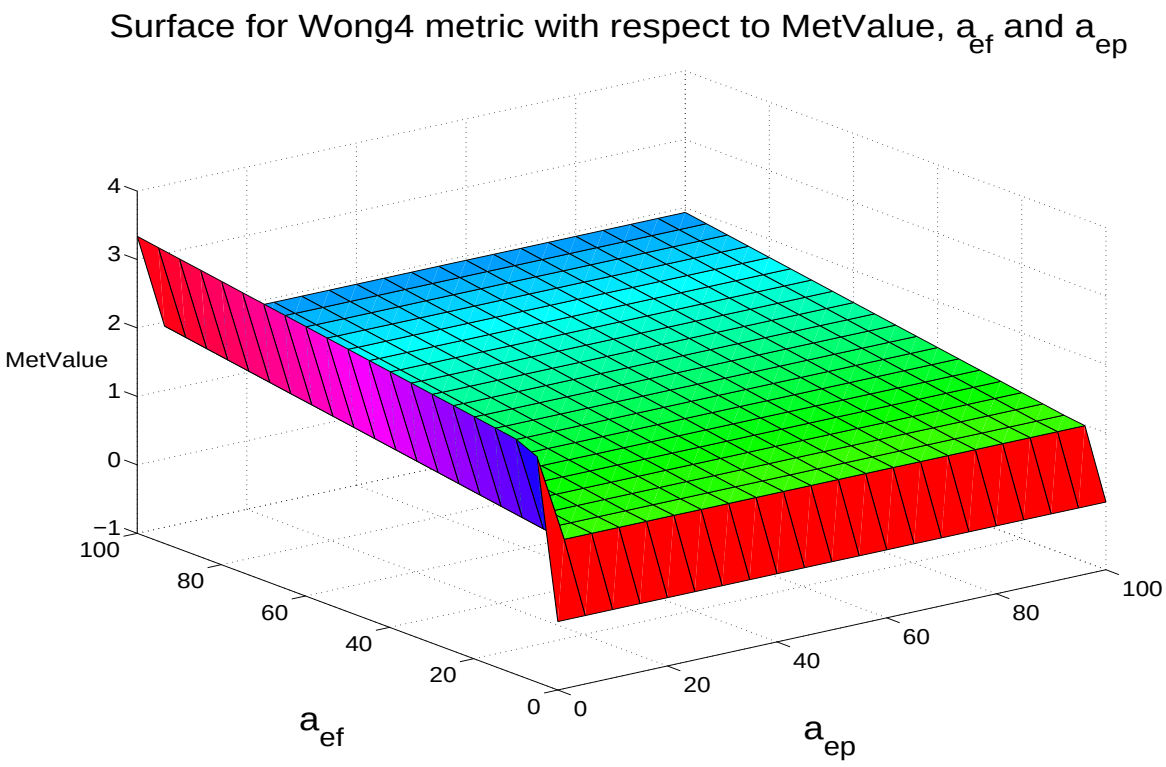

Figure 5.8: Surface for Wong4 metric

The Kulczynski2 metric of Figure 5.9 shows a very similar surface to the Wong3 and O^p metrics. However, the surface slants lower with a curve peak at 5 on the y-axis, a_{ef}, before converging to the metric value, $MetValue$, of 0 (a_{ef} of 0). Figure 5.10 shows the surface for the Tarantula metric. This surface is quite similar to Kulczynski2 except that the peak of the surface is at 1 on the y-axis, a_{ef} of 1 when the x-axis, a_{ep}, is 0. This is due to the design of the Tarantula metric where the numerator consists of a_{ef} and part of the denominator consists of a_{ep}. Another difference between this surface and the surface for Kulczynski2 is that the former surface slants lower as the a_{ep} increases and the a_{ef} is at the maximum (100).

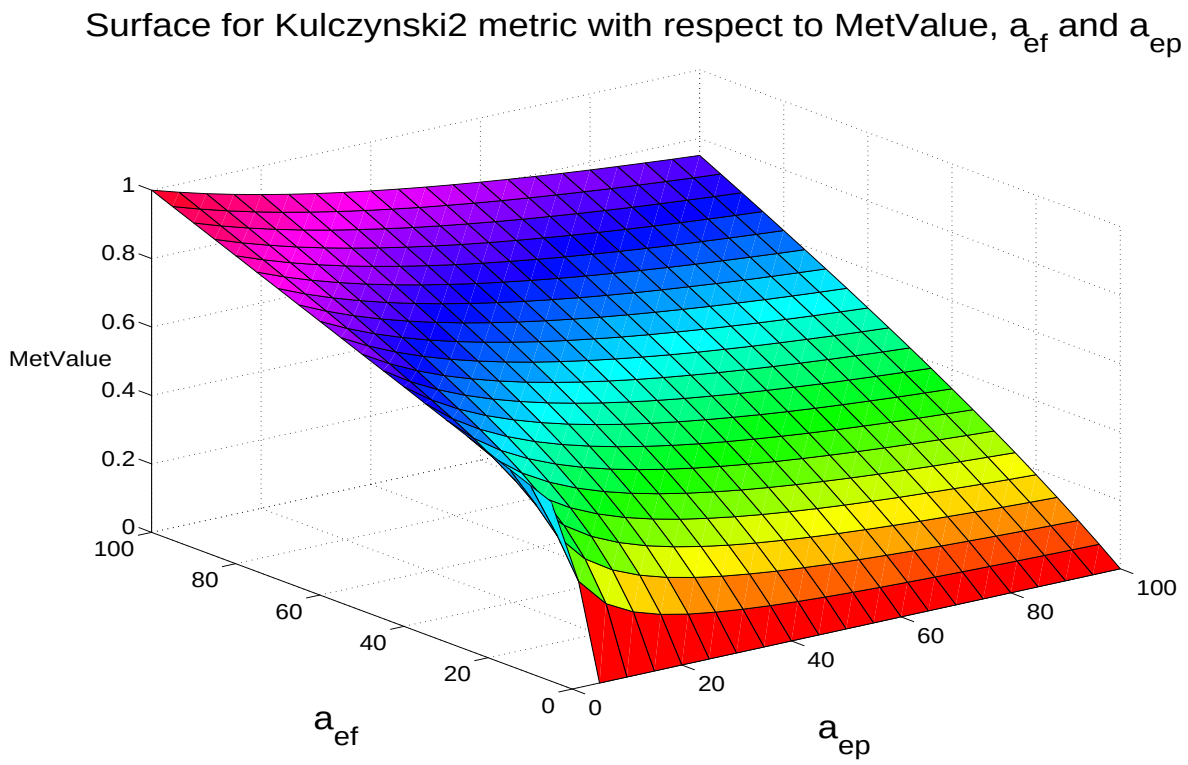

Figure 5.9: Surface for Kulczynski2 metric

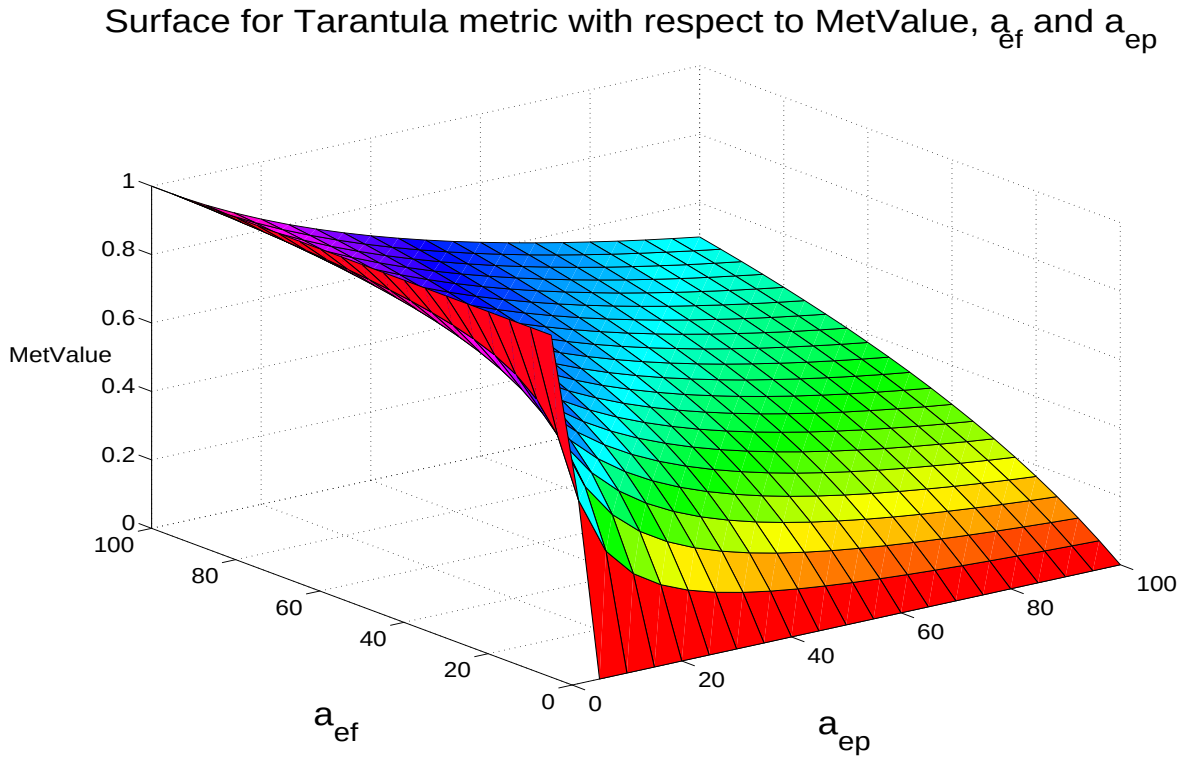

Figure 5.10: Surface for Tarantula metric

Figure 5.11 shows the surface for the Rogers metric, which slants lower than other metrics' surfaces. The surfaces for other metrics can be found in Appendix B.

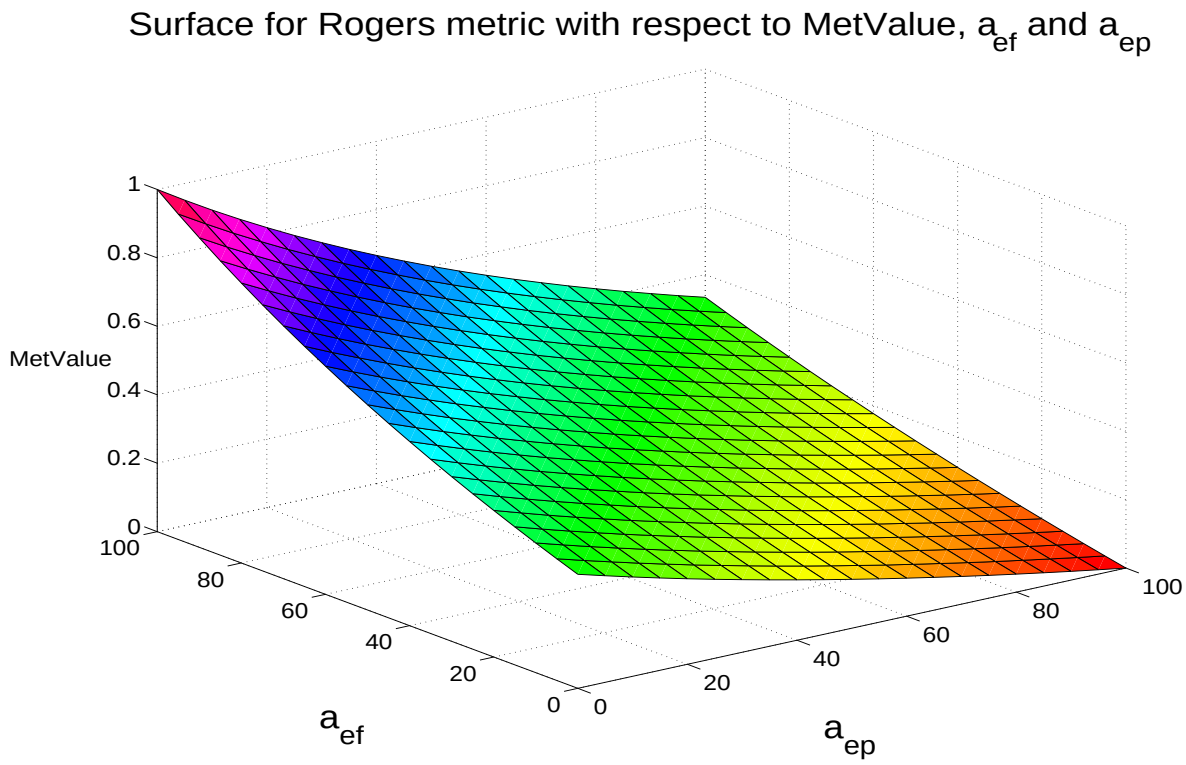

Figure 5.11: Surface for Rogers metric

In conclusion, all the surfaces of these metrics share an identical characteristic. The points on these surfaces peak when the a_{ef} and a_{ep} are at the maximum and minimum points respectively (all the fail test cases execute the particular statement but no pass test cases execute the statement).

5.7 Results Using Model Program

There are several reasons why our approach of modelling the diagnosis problem using a simple program such as $ITE2_8$ is beneficial. First, it is simple to run many experiments where various parameters are controlled precisely. The vagaries, biases, restrictions and cost of constructing or gathering *real* code and test suites are avoided. Such experiments can inspire hypotheses about the behaviour of various metrics etc., which can then be investigated further. For example, we have proved the equivalence of various metrics having noticed they produced the same results for all test suite sizes using our model — see Subsection 5.2.1. Second, if some result holds for the model and all the benchmarks we used for evaluation, say, we can be more confident it will also hold for other situations

(more than if the result holds for only one of these at least). Third, a model allows a more analytical approach, rather than a purely empirical one. A clear example is our development of optimal metrics.

5.7.1 Test Suite Size

We first investigate how well different spectra metrics perform with different numbers of tests using the $ITE2_8$ model program. We give the percentage score for each of the metrics discussed earlier for a variety of test suite sizes — see Table 5.1 and Table 5.2 respectively.

To compute these figures, we considered every multiset for up to 50 tests and sampled 2.1 billion multisets for larger numbers of tests. For 50 tests, the number of distinct multisets is around 264 million, and for 1000 tests it is around 2×10^{17}. Except where noted, we get consistent results to four decimal figures. The metrics are ordered according to the performance for 100 tests. We use the same ordering for the subsequent tables of results for the $ITE2_8$ model program. Several sets of metrics give the same results in all cases (we prove they are equivalent for ranking with respect to monotonically increasing functions – see Lemma 5.2.1 in Subsection 5.2.1). The first set (other than the O and O^p metrics, which we have already discussed) is Jaccard, Anderberg, Sneath & Sokal 2, Sørensen-Dice, Dice, Goodman, Levandowsky, and Kulczynski1, which we shall refer to collectively as "Jaccard". The second is Rogers and Tanimoto, Gower1, Gower2, Simple Matching, SokalDist, Hamann, Sneath & Sokal 1, M1, M3, Wong2, Euclid, Hamming, NFD, Manhattan, and Lee, which we label collectively as "Rogers". The third set, collectively labelled as "Scott", is Scott and Rogot1. The fourth set, collectively labelled as "Ochiai" is Ochiai, Ochiai3, and CorRatio. The fifth set, labelled as "Tarantula", is Tarantula, CBI Increase (CBI Inc), q_e, and Coef. The sixth set, collectively labelled "Braun", is Braun, Interest, and Forbes. This is followed by Russell and Rao, Simpson, Wong1, and Binary (for single-bug programs), which we shall collectively refer to as "Russell". Finally, the last set, collectively labelled "Pearson", is Pearson and Phi. We shall use the collective naming of these metrics throughout the thesis.

Due to the large number of metrics, we break down these metrics into two groups so as to facilitate readability; *Group A* (see Table 5.1), and *Group B* (see Table 5.2). We evaluate all these metrics using average rank percentages in Subsection 5.8.1, and observe that not all the metrics perform well in terms of bug localization performance. Therefore, for *Group A*, we choose several better performing metrics, especially those used in the debugging area, plus a set of representative metrics of moderate performance. The remaining metrics, which do not perform well in bug localization, are in *Group B*.

Table 5.1: Influence of Test Suite Size and *Group A* metrics on Total Score (%) for $ITE2_8$

Num. of tests	2	5	10	20	50	100	500	1000
O, O^p	60.00	72.59	81.10	87.98	94.14	96.81	99.32	99.64
Wong3′	60.00	65.74	76.38	86.86	94.07	96.81	99.32	99.64
Wong3	56.67	63.15	75.59	86.79	94.07	96.81	99.32	99.64
Zoltar	60.00	71.85	80.05	87.54	94.08	96.80	99.32	99.64
Wong4	60.00	66.30	71.85	73.23	86.80	94.55	99.22	99.59
JacCube	60.00	72.59	81.08	87.46	92.48	94.34	95.93	96.05
M2	60.00	72.59	80.99	87.37	92.42	94.31	95.89	96.03
Kulczynski2	60.00	71.85	79.90	86.34	91.79	93.93	95.80	95.97
McCon	60.00	71.85	79.90	86.34	91.79	93.93	95.80	95.97
Russell	53.33	61.11	69.57	78.79	88.91	93.83	98.64	99.30
Overlap	48.89	54.88	66.57	77.91	88.80	93.81	98.64	99.30
Ochiai	60.00	71.85	79.28	84.95	89.53	91.23	92.79	92.90
Rogot2	56.67	67.78	77.10	83.46	88.42	90.22	91.86	91.96
Pearson	48.33	67.13	76.80	83.32	88.22	90.01	91.65	91.75
AMean	48.33	67.13	76.75	83.17	88.01	89.80	91.43	91.55
Ample2	56.67	67.78	76.35	82.74	87.84	89.65	91.33	91.44
Jaccard	60.00	71.85	78.63	83.38	87.09	88.53	89.91	89.99
Tarantula	60.00	66.30	71.60	76.42	80.58	82.21	83.77	83.91
CBI Log	25.00	49.32	63.52	73.12	78.68	80.23	82.40	82.88
Ample	36.67	40.37	44.94	42.88	44.75	46.36	45.98	45.92

Table 5.2: Influence of Test Suite Size and *Group B* metrics on Total Score (%) for $ITE2_8$

Num. of tests	2	5	10	20	50	100	500	1000
Conviction	43.33	66.85	78.67	87.30	94.07	96.80	99.32	99.64
YuleY	48.33	67.13	78.67	87.30	94.07	96.80	99.32	99.64
YuleQ	48.33	67.13	78.67	87.30	94.07	96.80	99.29	99.54
Gower3	48.33	67.13	78.67	87.30	94.07	96.80	99.29	99.54
Confidence	53.33	61.11	69.57	78.79	88.91	93.83	98.64	99.30
AssocDice	53.33	61.11	69.57	78.79	88.91	93.83	98.64	99.30
Kappa	43.33	66.48	76.16	83.81	90.08	92.65	94.97	95.22
Fager	60.00	72.59	80.67	86.45	90.78	92.23	93.28	93.24

Continued on next page

Table 5.2 – continued from previous page

Num. of tests	2	5	10	20	50	100	500	1000
Fossum	6.67	44.26	69.96	82.64	89.41	91.35	92.84	92.93
Klosgen	46.67	44.81	54.85	67.39	82.98	90.54	97.91	98.94
Mountford	60.00	71.11	78.32	83.89	88.43	90.15	91.73	91.85
HMean	48.33	67.13	77.06	83.50	88.43	90.22	91.86	91.96
GMean	48.33	67.13	76.80	83.32	88.22	90.01	91.65	91.75
Ochiai2	48.33	67.13	75.14	80.93	85.22	86.80	88.34	88.46
Cohen	48.33	67.13	75.16	80.67	84.75	86.27	87.76	87.87
Dennis	48.33	65.28	73.25	79.37	84.15	85.90	87.53	87.66
Baroni	60.00	70.00	75.94	79.88	83.00	84.23	85.45	85.53
Certainty	33.33	54.26	66.97	75.57	81.84	83.82	85.42	85.52
Braun	60.00	66.30	71.60	76.42	80.58	82.21	83.77	83.91
Fleiss	56.67	65.93	72.54	76.70	80.07	81.39	82.76	82.85
Scott	56.67	66.67	72.39	76.46	79.64	80.87	82.17	82.27
CBI Sqrt	25.00	46.30	60.67	70.99	77.14	78.61	79.65	79.72
Rogers	56.67	63.15	67.60	71.02	73.89	75.07	76.32	76.42
CollectiveS	10.00	8.89	34.06	57.35	71.31	74.61	76.41	76.48
YuleV	26.67	52.41	64.25	68.85	72.60	73.96	75.28	75.42
AddedValue	13.33	34.44	45.65	52.75	57.77	59.50	61.19	61.31
J-Meas	33.33	28.89	29.67	31.71	34.69	35.92	37.18	37.30
Platetsky-Shapiro	46.67	39.44	33.69	29.59	26.80	25.85	25.17	25.11
Correlation	33.33	7.59	5.06	2.65	1.48	1.14	0.89	0.85

Table 5.1 shows that the optimal metrics, O and O^p, perform equal to or better than all other metrics for all the numbers of tests considered. Since more tests provide more information, we would expect performance to increase as the number of tests grows. This is born out in our results for all the *Group A* metrics except Ample. With some metrics, such as Rogers and Tanimoto, and Jaccard metrics, the performance grows relatively slow. With others, such as Overlap, it grows more quickly. As the number of tests increases, the JacCube metric outperforms the Jaccard metric. Ample performs particularly poorly for our model. This is because it always ranks $S4$ and $S3$ equally (since $S4$ is executed if and only if $S3$ is not executed). Thus, the maximum score the Ample metric can achieve is 0.5 rather than 1. These results are reasonably consistent with previous empirical studies comparing Ochiai, Jaccard, Tarantula, and Ample for software fault diagnosis [Abreu

et al., 2007] and Ochiai, Simple Matching, and Rogers and Tanimoto for computing genetic similarity in molecular biology [Meyer et al., 2004].

Table 5.2 shows the performance for *Group B* metrics with respect to the different numbers of tests. As the number of tests grow, we also observe the performance increases for all the *Group B* metrics except Fager, Klosgen, CollectiveS, J-Meas, Platetsky-Shapiro, and Correlation. As the number of tests grows beyond 5, by using the top-rank-bug score, Correlation metric performs worse than the random guess of buggy statement in the $ITE2_8$. Such observation is not observed using rank percentages measure in Table 5.12. Metrics in Table 5.2 do not show any better performance as compared to the better performing metrics in *Group A* table. Except for Table 5.12, we report the result of the *Group A* metrics throughout the thesis to ease readability.

5.7.2 Error Detection Accuracy

Error detection accuracy, q_e [Abreu et al., 2006], is defined as the proportion of fail tests in test cases where the buggy statement was executed (Definition 18).

Definition 18 (Error detection accuracy, q_e)**.**

$$q_e = \frac{a_{ef}}{a_{ef} + a_{ep}}$$

It is an indication of how *consistent* a bug is, though it depends on the test set used. Some bugs always result in failure when they are executed (called *deterministic* bugs in [Liblit et al., 2005]), whereas some return the correct result for nearly all tests in which they are executed. We would expect less consistent bugs to be harder to diagnose. This has been studied empirically [Abreu et al., 2006, Abreu et al., 2007, Lee et al., 2009a], and we discuss extensively in Chapter 6. Here, we study briefly using our model.

Table 5.3 gives the performance of the metrics for 100 test cases and several ranges of q_e values. For this (and subsequent) tables, we use strict inequality for the lower bound and non-strict for the upper bound. For example, 0.9–1.0 means $0.9 < q_e \leq 1.0$. We generate multisets of paths as before, and for each one, we determine the q_e value. The second row of the tables gives the percentage of multisets for which the q_e value is in the range; this peaks around 0.5. Multisets where there are no fail tests are not in any of the ranges, so the total percentage is slightly less than 100. The metrics in Table 5.3 are sorted according to the q_e range, 0.2–0.5, as our empirical benchmarks in Section 5.8 have an average q_e value for all benchmark programs of approximately 0.3796.

Table 5.3: Influence of Error Detection Accuracy, q_e, with 100 tests for $ITE2_8$

q_e **range**	**0.00-0.05**	**0.05-0.10**	**0.1-0.2**	**0.2-0.5**	**0.5-0.9**	**0.9-1.0**
% of cases	1.06	2.48	8.03	38.91	45.38	3.77
O, O^p	65.47	83.12	91.09	96.68	99.16	99.94
Wong3′	65.45	83.11	91.08	96.67	99.15	99.93
Wong3	65.45	83.11	91.08	96.67	99.15	99.93
Zoltar	65.47	83.11	91.08	96.67	99.15	99.94
Wong4	48.09	46.04	82.96	96.23	98.57	99.37
Russell	59.50	76.59	86.68	93.68	96.62	97.33
Overlap	59.49	76.59	86.67	93.67	96.60	97.24
JacCube	64.69	79.68	85.91	92.86	98.19	99.93
M2	65.05	80.35	86.38	92.79	98.04	99.93
McCon	65.44	82.59	88.58	92.28	97.14	99.90
Kulczynski2	65.44	82.59	88.57	92.28	97.14	99.90
Ochiai	63.09	75.64	80.74	88.34	96.48	99.90
Rogot2	62.34	74.39	79.20	86.80	95.94	99.89
Pearson	61.98	73.80	78.72	86.59	95.79	99.88
Ample2	62.48	74.70	79.61	86.50	94.92	99.74
AMean	61.61	73.20	78.23	86.37	95.64	99.87
Jaccard	58.59	67.83	73.28	84.41	95.77	99.90
CBI Log	40.88	63.47	69.27	77.04	85.90	92.25
Tarantula	58.05	65.97	69.29	76.90	89.27	98.82
Ample	31.42	39.05	39.73	44.57	49.47	51.99

The performance of all metrics in the table significantly increases as q_e increases. Although our optimality result does not apply to limited q_e ranges, the optimal metrics perform better than all other metrics in all the cases observed. The margin between the best metrics and poorer ones is greatest for small q_e values. There are a few notable differences in relative performance compared with Table 5.1. First, despite good overall performance, Overlap and Russell perform relatively poor for very low and very high q_e values. Second, for low q_e, Ample2 performs relatively well, and Kulczynski2 performs better than M2.

5.7.3 The Number of Fail Tests

The number of fail tests can also affects diagnosis performance, which is what we investigate next. Table 5.4 gives the performance of the metrics with 100 test cases and various ranges for the proportion of tests that failed. Due to the very small percentage of multi-sets, figures in the last column may be inaccurate in the last two digits. The metrics in the Table 5.4 are sorted according to the smallest proportion of fail tests range, 0.1–0.2, as our empirical benchmark in Section 5.8 has an average fail proportion value for all the benchmark programs of approximately 0.1980.

Table 5.4: Influence of Proportion of Fail Tests with 100 tests for $ITE2_8$

Failure range	**0.00-0.05**	**0.05-0.10**	**0.1-0.2**	**0.2-0.5**	**0.5-0.9**	**0.9-1.0**
% of cases	6.25	11.02	26.54	49.26	6.55	0.002
O, O^p	86.86	94.24	96.79	98.36	99.11	99.41
Wong3′	86.85	94.23	96.79	98.35	99.10	99.11
Wong3	86.85	94.23	96.79	98.35	99.10	99.11
Zoltar	86.86	94.23	96.79	98.35	99.09	99.01
Wong4	66.87	89.23	96.48	97.84	97.88	90.79
McCon	86.76	93.18	94.16	94.67	96.07	98.42
Kulczynski2	86.76	93.18	94.15	94.67	96.07	98.42
M2	86.34	92.13	93.93	95.65	97.42	99.21
Russell	74.36	88.76	93.76	96.82	98.31	98.51
Overlap	74.36	88.76	93.76	96.80	98.23	94.36
JacCube	85.90	91.61	93.76	95.94	97.86	99.21
Ochiai	84.30	88.74	90.45	92.65	95.46	98.42
Ample2	84.58	88.95	89.96	90.38	90.10	85.54
Rogot2	84.25	88.27	89.48	91.37	94.48	96.24
Pearson	83.64	87.80	89.34	91.28	93.85	95.64
AMean	83.04	87.34	89.21	91.18	93.27	94.65
Jaccard	80.80	84.43	86.86	90.61	94.85	98.42
CBI Log	75.60	82.47	82.95	79.72	74.90	69.70
Tarantula	79.97	82.14	82.54	82.53	81.80	75.64
Ample	43.92	44.51	46.59	47.08	46.24	50.79

Overall, these results show patterns for the top five metrics are very similar to those in Table 5.3, with the optimal metrics performing best in all cases. This is to be expected

since q_e is related to the proportion of failures. However, the results do show that some metrics, such as Tarantula and Ample, are less sensitive to the proportion of failures than others. Also, Tarantula, its equivalent CBI Increase (CBI Inc), and the variant CBI Log, peak at around 0.1 to 0.5, rather than monotonically increasing. Performance of Overlap and Ample2 also decrease slightly for high proportions of failures.

In a previous study [Abreu et al., 2007], there is empirical evidence that around 6 fail test cases is sufficient for good performance of Ochiai. Increasing the number of pass tests beyond 20 only resulted in a minor improvement in performance. We reproduce this experiment within the constraints of 4, 5, and 6 fail tests, on all the metrics. In our implementation, we generate multiple random numbers for the multisets for large number of fail tests. This implementation can reduce the time to produce multisets within the constrained number of fail tests. The results of 4, 5, and 6 fail tests are consistent with those of [Abreu et al., 2007] — most metrics show only a modest performance increase. The performance of a few of the poorer metrics actually decreases with more pass tests. Since the ordering of most metrics are similar for 4, 5, and 6 fail tests, we only show the result for 5 fail tests in Table 5.5. As with Table 5.4, when the number of tests is large (and thus the proportion of fail tests is small), Ample2 performs well. The Russell and Overlap metrics perform relatively poorly, and Kulczynski2 performs better than M2.

Table 5.5: Influence of Pass Tests with 5 Fail Tests for $ITE2_8$

Num. of tests	**10**	**20**	**50**	**100**	**500**	**1000**
O, O^p	88.91	90.48	91.22	91.47	91.62	91.65
Wong3′	83.87	89.45	91.15	91.46	91.62	91.65
Wong3	83.87	89.45	91.15	91.46	91.62	91.65
Zoltar	87.17	90.00	91.18	91.46	91.62	91.65
Kulczynski2	86.53	88.49	90.45	91.21	91.61	91.65
McCon	86.53	88.49	90.46	91.21	91.61	91.65
M2	88.44	89.68	90.15	90.30	90.37	90.39
JacCube	88.76	89.66	89.77	89.67	89.50	89.49
Ample2	79.00	84.92	87.07	87.68	88.08	88.15
Rogot2	81.50	84.64	86.42	87.13	87.91	88.06
Ochiai	85.49	86.38	87.05	87.27	87.41	87.45
Pearson	81.56	84.87	86.07	86.44	86.69	86.74
AMean	81.56	84.86	85.72	85.78	85.66	85.66
Russell	83.34	83.33	83.32	83.35	83.33	83.32
Overlap	77.83	82.74	83.29	83.35	83.33	83.32

Continued on next page

Table 5.5 – continued from previous page

Num. of tests	10	20	50	100	500	1000
Jaccard	84.89	84.27	83.33	82.87	82.40	82.37
CBI Log	72.44	79.08	81.31	81.96	82.37	82.42
Tarantula	70.28	77.50	80.62	81.50	82.13	82.23
Wong4	78.03	72.40	66.84	64.70	62.90	62.74
Ample	39.55	42.46	43.54	43.84	44.04	44.07

5.7.4 Buggy Code Execution Frequency

If buggy code is rarely executed, bugs can be hard to diagnose. Here, we investigate the relationship between bug localization performance and the proportion of tests that execute the buggy statement ($S4$). To our knowledge, this has not been studied previously. Note that we only consider cases in which there is a fail test case, so $S4$ must be executed at least once. The number of executions of the buggy statement, error detection accuracy, and failure rate, are related: the number of failures is the number of tests in which $S4$ is executed times q_e. That is, the number of executions of $S4$ is the failure rate divided by q_e. The results are shown in Table 5.6, based on the same ordering of metrics in Table 5.1.

Table 5.6: Influence of Buggy Code Execution Frequency with 100 tests for $ITE2_8$

S4 exec range	0.00-0.05	0.05-0.10	0.1-0.2	0.2-0.5	0.5-0.9	0.9-1.0
% of cases	0.064	0.431	3.74	46.52	48.49	0.389
O, O^p	99.89	99.76	99.41	97.91	95.55	94.23
Wong3′	99.89	99.74	99.40	97.91	95.54	94.18
Wong3	99.89	99.74	99.40	97.91	95.54	94.18
Zoltar	99.89	99.75	99.41	97.90	95.54	94.07
Wong4	99.89	99.64	98.70	95.83	93.07	89.29
JacCube	99.89	99.76	99.39	97.30	91.23	80.36
M2	99.89	99.76	99.41	97.54	90.94	76.83
Kulczynski2	99.89	99.75	99.37	97.20	90.53	74.82
McCon	99.89	99.75	99.37	97.20	90.53	74.83
Russell	66.60	77.59	85.86	92.74	95.60	96.85
Overlap	66.59	77.58	85.85	92.73	95.58	96.61
Ochiai	99.89	99.74	99.30	96.31	85.99	64.76

Continued on next page

Table 5.6 – continued from previous page

S4 exec range	0.00-0.05	0.05-0.10	0.1-0.2	0.2-0.5	0.5-0.9	0.9-1.0
Rogot2	99.89	99.75	99.34	96.40	83.92	52.45
Pearson	99.89	99.74	99.29	96.15	83.77	46.38
AMean	99.89	99.73	99.23	95.90	83.63	41.19
Ample2	99.89	99.76	99.41	96.95	82.43	30.08
Jaccard	99.89	99.73	99.21	95.30	81.48	55.70
CBI Log	78.13	92.57	96.70	91.94	68.28	18.31
Tarantula	99.89	99.66	98.96	93.30	70.76	18.61
Ample	49.95	51.95	51.89	49.39	43.33	15.78

Perhaps surprisingly, metrics other than Russell, Overlap, the CBI Log, and Ample perform extremely well when $S4$ is executed only a few times, and performance decreases with greater execution frequency. For O and O^p metrics, the decrease is only slight. Russell and Overlap exhibit the opposite trend — performance increases as the frequency increases. For very high frequencies, these metrics perform better than O. This is the only case we have discovered where O does not perform the best. When $S4$ is executed in a large proportion of the tests, the a_{np} component of the O formula is misleading. It is typically very small for $S4$, but larger for $S1$ and $S2$. There are not many test cases for which a_{nf} is zero for $S1$ or $S2$, but there are enough to reduce the performance of O below that of Russell and also Overlap.

5.7.5 Comparison of Metrics

It would be naive to extrapolate results from our simple model to the diagnosis of real programs without further evidence. Nevertheless, our experiments suggest the following. The optimal metrics are best overall and are also quite robust. They perform best in nearly all the cases we have investigated. Zoltar performs almost as well as O in all cases and Wong3 (and Wong3′) performs similarly when the numbers of tests is reasonably large. Although Russell and Overlap are next in overall performance for larger number of tests, and can perform better than O when the bug is executed in a large proportion of the tests, they are less robust. For small numbers of tests, q_e values and proportion of fail tests results in their performance being significantly worse. In contrast, although M2 and Kulczynski2 perform somewhat worse than these metrics overall for large numbers of tests, they are more robust and our experiments suggest they may be better metrics to use in practice (and for small q_e and proportion of fail tests, Kulczynski2 is the better

of the two). Several of the other metrics also perform very well in some circumstances. However, even the ones with higher profiles, such as Tarantula, Jaccard, and Ochiai, do not have particularly good overall performance. When they perform very well, they are still not as good as the best metrics. A few metrics perform particularly poorly.

The reason why the Russell, Binary, Wong1, and Overlap metrics behave somewhat differently from the other metrics, can be understood by considering the two conditions for optimality. Recall metrics are optimal if under these conditions.

1. they increase as a_{np} increases when $a_{nf} = 0$
2. a_{np} always has smaller values when $a_{nf} > 0$

When $a_{nf} = 0$, the Russell and Overlap metrics are minimal but constant, rather than increasing in a_{np} (satisfying the second condition but not the first one). All the other non-optimal metrics we consider are increasing in a_{np} when $a_{nf} = 0$ (this is straightforward to prove from the formulas), but do not always have smaller values when $a_{nf} > 0$ (satisfying the first condition but not the second condition). Further insight into the relative performance of the metrics can be obtained by constructing formulas that are equivalent to the metrics for ranking purposes but are easier to compare. Specifically, we can compare with formulas that are known to be optimal. In Table 5.7, we provide formulas that (for ranking) are equivalent to a selection of the formulas given earlier. A simpler terminology (similarly defined in Subsection 5.2.1) is used for some of the terms that have been defined in the Glossary.

Proposition 5.7.1. *Table 5.7 gives metrics that, for ranking, are equivalent to the definitions of metrics in Table 2.3, assuming a single bug and our top-rank-bug score in the case of Binary.*

Proof. The first optimal metric equivalent formula is O^p, given earlier. The second optimal formula is equivalent to $a_{ef}/(a_{ep} + P \cdot F)$. If $a_{nf} = 0$, this equals to $F/(a_{ep} + P \cdot F)$, which increases as a_{ep} decreases and is greater than the value of the metric for $a_{nf} > 0$, since the denominator is between $P \cdot F$ and $P \cdot (F + 1)$. Note that the term $P \cdot F$ can be replaced by any larger term. The rest are either given by the previous propositions or are straightforward. □

O^p is similar to the Wong3 metric, but does not have three separate cases with different a_{ep} coefficients: one very small coefficient (its maximum size dependent on the number of pass tests) is optimal and the other cases just decrease performance in the model. Note that the smallest a_{ep} coefficient in the Wong3 metric is also sub-optimal for more than 1000 tests. With larger numbers of test cases, we see the performance of the Wong3

Table 5.7: Equivalent Formulas for the Spectra Metrics used

Metric(s)	Equivalent Formula
Optimal (O^p)	$a_{ef} - \frac{1}{P+1}a_{ep}$
Russell	a_{ef}
Rogers	$a_{ef} - a_{ep}$
Ample2	$a_{ef} - \frac{F}{P}a_{ep}$
Ample	$\left\|a_{ef} - \frac{F}{P}a_{ep}\right\|$
Kulczynski2	$a_{ef} - \frac{F}{a_{ep}+a_{ef}}a_{ep}$
Optimal	$\log a_{ef} - \log\left(a_{ep} + P.F\right)$
Tarantula	$\log a_{ef} - \log a_{ep}$
Jaccard	$\log a_{ef} - \log\left(a_{ep} + F\right)$
JacCube	$\log a_{ef} - \frac{1}{3}\log\left(a_{ep} + F\right)$
M2	$\log a_{ef} - \log\left(a_{ep} + 2F + P\right)$
Ochiai	$\log a_{ef} - \frac{1}{2}\log\left(a_{ep} + a_{ef}\right)$

metric again drops noticeably below that of O and Zoltar metrics. For 5000 tests, the O and Zoltar metrics have a percentage score of 99.93, compared with 99.91 for Wong3 metric.

Comparing the O^p formula with those for the Rogers and Russell metrics, we see that the Russell metric neglects the a_{ep} term, whereas Rogers gives it too much influence. For a large number of pass tests, this means the performance of the Russell metric approaches to that of O^p (since the optimal coefficient of a_{ep} approaches zero). Conversely, Rogers is closer to the O^p for very small number of pass tests. For large number of tests, Rogers is one of the worst metrics, because of its relatively large coefficient for a_{ep}. It is particularly bad when the failure rate, q_e, or $S4$ are small (since a_{ep} is relatively large compared to a_{ef} in these cases).

The performance of the Ample2 metric is particularly sensitive to the failure rate. For a single fail test, it is almost as good as O^p (the a_{ep} coefficient is $1/(P+1)$) but when the failure rate is more than 50%, it is even worse than the Rogers metric (the a_{ep} coefficient is more than 1). In practice, failure rates are typically quite small, helping the performance of Ample2. The use of absolute value in the Ample metric makes it less like O^p; this is what prompted us to experiment with the Ample2 metric, which indeed outperforms Ample. Kulczynski2 performs better than Ample2, particularly for a high number of failures, for the following reason. The Kulczynski2 a_{ep} coefficient is similar to that of Ample2, but the denominator is the number of tests in which the statement was executed,

rather than P. For the buggy statement, this will be between F and $F + P$, whereas for correct statements it will tend to be lower, lowering the rank of these statements.

Among the formulas with logarithms, it can be seen that M2 is better than Jaccard, which is better than Tarantula due to the relative influence of a_{ep}. The Jaccard metric gets close to optimal for very high failure rates, and M2 gets close to optimal for both very high and very low failure rates. The JacCube metric (which we modify based on Jaccard) is equivalent to reducing the influence of a_{ep} to the factor of $\frac{1}{3}$. Therefore, we observe that the performance using this metric is better than the Jaccard metric. For the Ochiai metric, the use of the square root (leading to the factor of $\frac{1}{2}$) is helpful to performance. Similarly to the Jaccard metric, a version of cube root for the Ochiai metric would be even better, as it would reduce the influence of a_{ep} even more.

We have not found a formula equivalent to Zoltar that is easily comparable with an optimal formula. When $a_{nf} = 0$, the Zoltar metric is $a_{ef}/(a_{ef} + a_{ep})$, which is increasing in a_{np} but may be smaller than some cases when $a_{nf} > 0$. For the buggy statement $S4$, the value is $F/(F + a_{ep})$. Due to the large factor of 10 000, it is rare for the value for $S1$ or $S2$ to be higher unless $a_{ep} = 0$. Even when $a_{ep} = 0$, we still need $a_{ef}/F > F/(F + a_{ep})$ ($S4$), which requires a low value of a_{nf} and a high value of a_{ep} for $S4$. That is, a correct statement ($S1$ or $S2$) is used in no pass tests but many fail tests, and the buggy statement is used in many pass tests. For example, with five tests, three of which fail with $S4$ used in both pass tests and $S1$ used in two fail tests and no pass tests, the Zoltar metric for $S1$ is (just) larger than that for $S4$ ($10/15$ compared with $9/15$). Such cases form only a tiny fraction of the multisets for larger number of tests.

5.7.6 Other Models

Many experiments with other models have been conducted. Here are some of the results briefly, which may be expanded on in the future. Increasing the number of paths while retaining the same code increases the number of possible multisets of test cases. This has little effect on the relative performance of the different metrics but does increase absolute performance. This is because a greater proportion of the multisets have a relatively even distribution across the different paths (for example, the proportion of multisets that contain just one path becomes much smaller). Increasing the number of sources of "noise" (adding extra correct if-then-else statements) with a fixed number of paths decreases performance, as expected. Decreasing the proportion of paths through the buggy statement that lead to failure (thus decreasing the average q_e value) also decreases performance. In all the cases above, O appears to be optimal.

We have discovered some classes of models where O is not optimal according to the definition we have given. If the buggy statement is (almost) always executed then metrics

such as the Russell metric can perform better than O. In Table 5.6, we have this result for just a subset of the multisets possible with $ITE2_8$, but by changing the model, it can occur with the *total score*. For such models, we believe it is appropriate to generalise our definition of optimality. Rather than fixing a specific statement as being buggy, we could consider the possibility of each statement being buggy, and average over all these cases. For $ITE2_8$, the two definitions are equivalent due to the symmetry of the code. With such a revised definition, O still appears optimal (for Russell, the performance is excellent when the frequently executed statement is buggy but poor when the statement is correct).

5.8 Results Using Empirical Benchmarks

We use the Siemens Test Suite, the subset of the Unix Test Suite, the Concordance, and the Space benchmarks to perform empirical evaluation on how well the metrics perform. Table 4.1 in Chapter 4 lists the programs, the number of faulty versions of each program, the number of lines of code (LOC), and the number of test cases. For the Space program, we evaluate *AllTests* set where the entire test suite of Space is used.

We conduct experiments using the same metrics as previously reported in our $ITE2_8$ model program. We report the figures for all the benchmarks; Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space. There are 130 Siemens Test Suite programs, 102 subset of the Unix Test Suite programs, 13 Concordance programs, and 28 Space programs that failed at least a fail test case. We also report on a subset of the test suites that allows a better comparison with our model. We use the set of programs that failed a test case and have a single bug. There are 122, 102, 11, and 15 single bug Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space programs respectively.

We provide experimental results using the different established performance measures of bug localization performance for the different metrics. One of them is *rank percentages* (Subsection 4.3.1) which refers to the percentages of the program code that needs to be examined for a bug to be found. The average of the percentages over all programs in the test suite are considered. We show the bug localization performance by considering different lines of code (as discussed in Section 4.2). For one set of results, we consider all the lines of code (Table 5.8). For the others, we ignore lines that are not executed in any tests.

5.8.1 Average Rank Percentages

In this subsection, for ease of readability, some of the program names have been abbreviated. In the Siemens Test Suite, schedule is referred to as *Sch*, schedule2 is referred to as *Sch2*, print_tokens is referred to as *Pt*, print_tokens2 is referred to as *Pt2*, tot_info is referred to as *Tot* and replace is referred to as *Rep*. In the subset of the Unix Test Suite, Checkeq is referred to as *Chck* and Spline is referred to as *Spl*. Concordance is abbreviated as *Conc*.

Table 5.8: Average Rank Percentages for programs of the Siemens Test Suite

Metric	Tcas	Sch	Sch2	Pt	Pt2	Tot	Rep
O^p	9.90	3.88	23.21	3.40	0.78	3.11	4.68
O	9.90	7.65	23.21	3.40	0.78	3.11	4.68
Wong3′	10.11	3.88	17.35	3.40	0.82	3.11	3.95
Wong3	17.78	17.71	29.10	3.40	0.82	9.27	11.29
Wong4	10.03	4.74	20.77	3.55	0.78	3.18	4.68
Zoltar	9.90	3.88	20.77	3.40	0.78	3.13	3.92
M2	10.27	1.57	23.17	4.31	2.00	4.59	4.49
Kulczynski2	9.94	3.88	20.80	3.40	1.08	3.27	4.28
Overlap	14.47	11.36	18.51	7.15	10.65	6.63	9.11
Ochiai	10.66	1.63	23.56	6.22	3.70	5.52	4.90
Amean	10.82	5.14	23.89	7.31	5.09	9.37	5.24
Jaccard	10.77	1.68	26.04	8.37	5.55	6.53	6.25
Tarantula	10.80	1.77	26.04	8.93	5.70	7.09	6.45
Russell	14.47	11.36	18.51	7.15	10.65	6.60	9.10
Binary	14.47	15.15	18.51	7.15	10.65	6.60	9.10
Ample	12.83	12.62	27.50	8.32	5.47	15.09	6.92
Ample2	10.91	4.98	28.42	7.71	4.94	9.38	5.72
Pearson	10.92	4.95	25.13	7.84	4.92	9.31	5.05
McCon	9.94	10.94	20.80	3.40	1.08	3.27	4.28
CBI Log	11.47	7.20	26.04	8.88	5.70	10.26	6.25
JacCube	10.35	2.08	22.91	4.57	2.00	4.53	4.58
Rogot2	10.91	10.83	27.76	7.74	4.92	14.37	5.80

Table 5.8 shows the average of rank percentages, calculated using all lines of program code, with respect to the different programs of the Siemens Test Suite, using different metrics. In this table, all the 132 programs of the Siemens Test Suite, including programs with multiple bugs and programs with no fail test, are included. The average rank percentages are computed based on the average of the rank percentages (Definition 11) of program versions of respective programs. In this table, we observe that the average rank percentages for the schedule (*Sch*) program evaluated using the O and O^p metrics

are different, at 3.88% and 7.65% respectively. The difference is due to one of the schedule (*Sch*) programs has multiple bugs. The bug localization performance evaluated using the O metric performs poorly in this program as the a_{nf} can be non-zero, especially for the buggy statements. This leads to the O metric being -1 for all the statements having non-zero a_{nf}.

Table 5.9: Average Rank Percentages for programs of the subset of the Unix Test Suite and Concordance

Metric	Col	Cal	Chck	Spl	Tr	Uniq	Conc
O^p	13.41	6.52	11.06	6.95	16.11	6.95	3.63
O	13.41	6.52	11.06	6.95	16.11	6.95	3.63
Wong3′	13.41	6.52	11.06	6.95	16.11	6.95	3.64
Wong3	13.41	6.52	11.06	11.69	16.11	6.95	25.79
Wong4	13.48	8.58	14.35	6.99	16.11	7.15	3.93
Zoltar	13.41	6.99	12.85	6.99	16.11	7.00	3.63
M2	13.41	7.10	13.45	6.99	17.97	8.95	3.84
Kulczynski2	13.41	7.57	14.38	7.04	16.71	9.00	3.66
Overlap	21.63	15.26	23.93	13.75	19.23	13.21	5.87
Ochiai	13.46	7.57	14.82	7.08	19.10	9.65	3.90
AMean	13.46	7.68	20.29	7.08	36.47	11.35	8.52
Jaccard	13.46	7.68	14.82	7.08	20.69	10.15	5.47
Tarantula	16.06	10.65	23.33	7.17	23.36	14.10	6.02
Russell	18.85	11.54	18.18	13.66	19.23	13.06	5.87
Binary	18.85	11.54	18.18	13.66	19.23	13.06	5.87
Ample	13.65	8.08	21.22	7.01	43.24	15.12	10.57
Ample2	13.41	7.68	20.84	6.99	36.54	12.50	8.29
Pearson	13.46	7.68	19.20	7.08	36.47	10.60	8.39
McCon	13.41	7.57	14.38	7.04	16.71	9.00	3.66
CBI Log	14.35	12.88	22.32	10.09	37.74	14.00	12.18
JacCube	13.41	7.10	12.91	7.04	17.97	8.90	3.71
Rogot2	13.52	7.84	17.38	7.04	55.5	10.15	22.70

Table 5.9 shows the different programs of the subset of the Unix Test Suite and Concordance (*Conc*) using different metrics, calculated using all lines of source code. For the Concordance dataset, we evaluate all 13 programs (two of which are multiple-bug programs). From Table 5.8 and Table 5.9, the O^p metric performs among the best across all programs, except for the schedule (*Sch*), schedule2 (*Sch2*), and replace (*Rep*) programs. For the latter programs, O^p metric does not perform well due to the inclusion of some programs in $schedule$, $schedule2$ and $replace$ that have no fail test cases or contain multiple bugs. In Section 5.5, we showed that O^p metric is an optimal metric for single bug programs. This metric does not perform the best for multiple-bug programs.

Table 5.10: Average Rank Percentages for all Single Bug Datasets

Benchmark	Siemens	Unix	Conc	Space	All
O	5.81	10.36	2.83	0.49	7.22
O^p	5.81	10.36	2.83	0.49	7.22
Wong3′	6.05	10.36	2.84	0.49	7.33
Zoltar	5.82	10.77	2.83	0.55	7.39
Kulczynski2	5.97	11.49	2.86	0.62	7.76
McCon	5.97	11.49	2.86	0.62	7.76
Wong4	6.05	11.36	3.18	0.49	7.76
JacCube	6.74	11.27	2.92	0.58	8.05
M2	6.72	11.36	3.07	0.58	8.08
Ochiai	7.38	11.93	3.13	0.67	8.65
Jaccard	8.59	12.19	4.96	1.06	9.45
Pearson	8.89	14.73	8.43	0.82	10.77
Amean	8.79	15.02	8.57	0.92	10.85
Ample2	8.94	15.26	8.33	0.82	11.00
Tarantula	8.81	15.77	5.61	1.88	11.09
Wong3	12.25	10.96	22.49	0.49	11.47
Russell	10.45	16.02	5.88	5.35	12.22
Binary	10.45	16.02	5.88	5.35	12.22
Rogot2	10.28	16.44	11.93	0.84	12.30
CBI Log	10.03	17.43	12.89	1.81	12.68
Overlap	10.46	18.49	5.88	5.35	13.23
Ample	12.14	16.55	10.98	2.28	13.30

We also consolidate all the figures of the single bug programs (including all the lines of code that are not executed), with respect to the benchmarks, in Table 5.10. Note that we also combine the evaluation of all the benchmarks in the last column of the table (All). As there are different proportions across the benchmarks, comparisons between these four columns should be avoided. In each dataset, we show that the O and O^p metrics perform the best or equal best to some other metrics. In the Siemens Test Suite, by using the O and O^p metrics, the programmer only needs to examine 5.81% of the program code in order to locate the bug. For the subset of the Unix Test Suite, Concordance, and Space programs, O and O^p metrics are among the best metrics, with the programmer only needing to examine 10.36%, 2.83%, and 0.49% respectively of the program code in order to locate the bug.

Considering different lines of code (see Section 4.2) can have a significant effect on the relative bug localization performance. This is what prompted us to introduce the Wong3′ metric (in several programs, the Wong3 metric as defined in [Wong et al., 2007] ranks the buggy statement below code that is never executed in any test case). In Table 5.10, we observe different bug localization performance using the Wong3 and Wong3′ metrics.

This is due to the buggy statement being ranked below code that is never executed in any test case for some of the programs.

χSuds [Telcordia Technologies, Inc., 1998] is used to extract the spectra and this apparently filters out some information `gcov` reports, leading to different results [Wong et al., 2007]. Our model program can be modified to include statements that are never executed, or debugging systems could *a priori* exclude such statements from consideration (making Wong3 and Wong3$'$ equivalent). We show bug localization performance considering only the lines of code that are executed in Table 5.11 and Table 5.12.

In Table 5.11, we evaluate using *Group A* metrics (the same metric selections as Table 5.1) on the single bug programs. We consolidate all the figures (using just lines of code that are executed) with respect to the benchmarks, namely the Siemens Test Suite (Siemens), subset of the Unix Test Suite (Unix), Concordance (*Conc*), and Space. The ordering of the metrics in this table is based on the combination of the bug localization performance for all these benchmarks (column All). We observe that the O and O^p metrics perform the best or equal best with several other metrics in bug localization performance with 15.69%, 20.47%, 10.11%, and 1.60% of the program code to be examined in order to locate single bug programs in the Siemens Test Suite, subset of the Unix Test Suite, Concordance (*Conc*), and Space benchmarks, respectively.

We also observe that the relative performance of the top performing metrics on the single bug programs of all the test suites (Table 5.11) fits reasonably well with the overall results of our model, with the optimal metrics performing best for all single bug programs. The top four ranked metrics (treating O and O^p as the same metric, and similarly Wong3 and Wong3$'$) in Table 5.11 are *identical* (modulo equal ranking) to those ranked according to the q_e range of 0.2-0.5 in Table 5.3 and the proportion of fail tests range of 0.1-0.2 in Table 5.4.

The most significant difference in results between our empirical benchmarks and the model program $ITE2_8$ is that the Russell, Binary, and Overlap metrics perform more poorly than the overall performance in our model. Performance of the Russell, Binary, and Overlap metrics is significantly affected by statements that are executed in all test cases, such as initialisation code. These are always ranked equal-highest. Adding such a statement in our model program approximately halves their model performance. The top-rank-bug score measure (Subsection 4.4.2) we use is also kind to these metrics, especially Binary. If Binary doesn't rank the bug equal-top, it ranks it equal-bottom. Other good metrics that fail to rank it equal-top will often rank it strictly above most correct statements, but the score will still be the same.

The JacCube metric is also slightly lower in the ranking than in our model. We observe that the metric has relative poor performance as compared to the McCon and Kulczynski2

Table 5.11: Average Rank Percentages for all Single Bug Datasets — executed lines of code only (*Group A* metrics)

Benchmark	Siemens	Unix	Conc	Space	All
O,O^p	15.69	20.47	10.11	1.60	16.55
Wong3	16.28	20.47	10.15	1.60	16.84
Zoltar	15.71	21.24	10.11	1.76	16.88
Wong4	16.27	22.11	11.35	1.60	17.56
Kulczynski2	16.13	22.58	10.21	2.01	17.65
McCon	16.13	22.58	10.21	2.01	17.65
JacCube	18.31	22.15	10.43	1.87	18.54
M2	18.28	22.32	10.97	1.86	18.62
Ochiai	20.17	23.37	11.19	2.12	20.00
Pearson	22.07	25.35	18.42	2.55	22.08
Jaccard	23.47	23.85	17.68	3.19	22.15
AMean	21.81	25.88	18.90	2.82	22.20
Ample2	22.21	26.33	18.05	2.54	22.53
Rogot2	23.84	26.14	19.25	2.59	23.30
CBI Log	24.46	29.59	22.63	6.01	25.37
Tarantula	24.12	30.65	20.03	6.22	25.53
Ample	31.39	28.69	27.53	6.62	28.63
Russell	28.36	31.99	21.03	17.30	28.85
Binary	28.36	31.99	21.03	17.30	28.85
Overlap	28.39	36.81	21.03	17.30	30.84

in the Column Siemens of the Table 5.11. This is expected, since they perform poorly for low q_e and proportion of fail tests. In the Siemens Test Suite, we observe that the q_e value and the fail proportion are 0.1276 and 0.035, respectively. In the range of 0.1-0.2 of Table 5.3 and 0.00-0.05 of Table 5.4, we observe JacCube has relatively poor performance as compared to the McCon and Kulczynski2 metrics. However, we do not observe the latter case in the subset of the Unix Test Suite and Space. There are many programs in these test suites that have ties between the buggy statement and other non-buggy statement(s), which can affect bug localization performance. We defer these details to Subsection 6.3.2 of Chapter 6.

Ample performs rather better than predicted by the model. If the bug occurs in an if-then-else statement, the Ample metric typically gives both the *then* and *else* statements the same reasonably high rank. Since this is still only a small fraction of the code (as opposed to half the code in our model) the performance overall is reasonable. Experiments using different models indicate that as the number of statements increases, the difference in performance between Ample and the better metrics decreases.

Another interesting observation is that there are 30 Siemens Test Suite programs, 17

Table 5.12: Average Rank Percentages for all Single Bug Datasets — executed lines of code only (*Group B* metrics)

Benchmark	Siemens	Unix	Conc	Space	All
Conviction	15.84	23.54	17.79	1.76	18.22
Fager	19.68	23.14	11.10	2.08	19.66
Fossum	20.46	23.48	18.27	2.12	20.50
Mountford	21.40	23.81	11.97	2.86	20.86
HMean	22.03	25.13	18.33	2.59	21.97
GMean	22.07	25.35	18.42	2.55	22.08
Certainty	23.80	25.59	19.43	5.49	23.24
Dennis	23.18	26.99	18.85	3.29	23.35
Kappa	22.62	26.64	18.22	10.65	23.35
Cohen	24.02	26.28	19.39	3.74	23.52
CBI Sqrt	24.50	29.42	22.89	5.97	25.32
Ochiai2	27.37	27.98	19.44	4.76	25.91
Braun	24.12	32.58	20.03	6.22	26.32
Baroni	30.68	26.57	21.05	7.79	27.21
Gower3	27.50	35.73	27.51	17.81	30.27
YuleQ	27.50	35.73	27.51	17.92	30.28
YuleY	27.50	35.73	27.51	17.92	30.28
AssocDice	28.39	36.81	21.03	17.92	30.87
Confidence	28.39	36.81	21.03	17.92	30.87
YuleV	31.17	36.69	21.22	13.33	31.91
AddedValue	26.79	45.75	23.77	35.37	34.91
Scott	41.93	29.26	41.23	15.83	35.16
Fleiss	42.23	29.39	41.45	16.11	35.39
Platetsky-Shapiro	44.47	30.31	49.03	24.93	37.72
Rogers	44.48	30.47	49.16	25.34	37.82
CollectiveS	44.50	30.66	49.31	25.34	37.92
J-Meas	47.62	33.22	34.45	30.28	40.13
Klosgen	41.06	43.73	31.96	32.99	41.26
Correlation	47.74	47.92	38.89	45.37	47.28

subset of the Unix Test Suite programs, and 4 Concordance programs, where Russell performs better than O, despite relatively poor performance overall. In all these cases, the buggy statement is executed on average 81% of the total test cases. This is consistent with our model (see Table 5.6) — when $S4$ is executed in 0.50–0.90 of tests, Russell outperforms O and is the best of all metrics considered.

We also evaluate *Group B* metrics on our benchmarks in Table 5.12. We apply the same ordering of the metrics in this table, based on the bug localization performance

across all benchmarks (Column All). However, we do not perform further analysis on the metrics in this group, as these metrics do not perform particularly well.

Figure 5.12 shows the plot of the breakdown of the average rank percentages (executed lines of code only) for the respective single bug (one-bug) programs in the Siemens Test Suite, the subset of the Unix Test Suite, and the Concordance (Conc) benchmarks, using the optimal metric O^p. This figure also shows the error bars of each program, which refers to the standard deviation of the average rank percentages for different program versions.

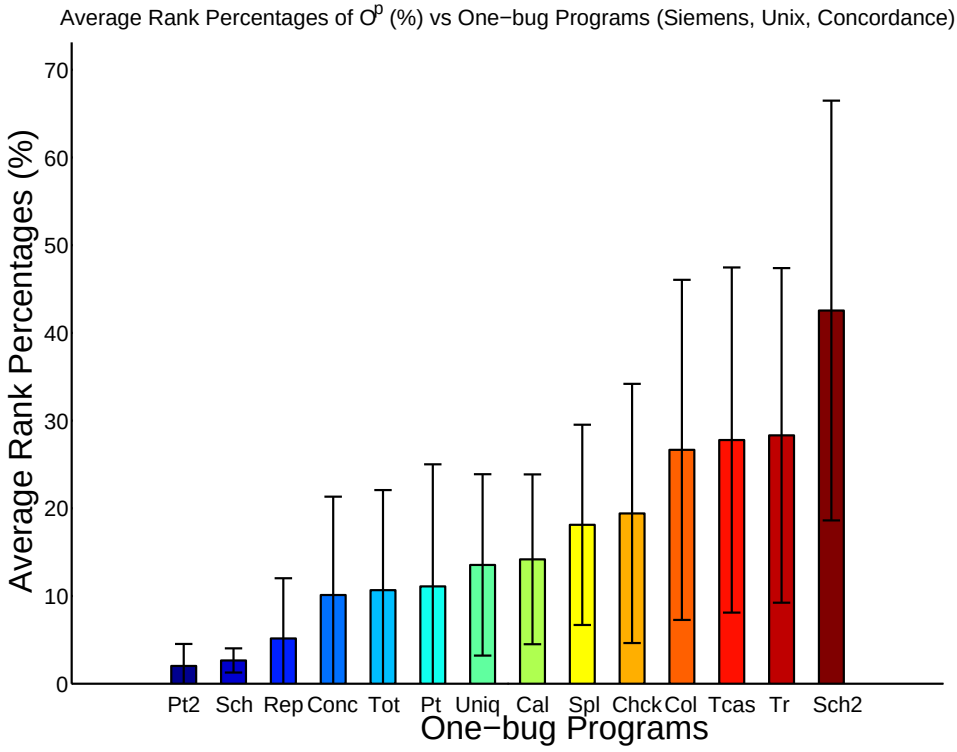

Figure 5.12: Average Rank Percentages with Error Bars evaluated using O^p metric on One-bug (Single Bug) Programs of Siemens Test Suite, subset of the Unix Test Suite, and Concordance (Conc)

In this figure, we observe that print_tokens2, *Pt2*, is the program that shows best bug localization performance (average rank percentages) evaluated using the O^p metric, with 2.02% of the program code to be examined by the programmer in order to locate the bug. On the other end of this figure, the schedule2, *Sch2*, program is the program that shows the worst bug localization performance evaluated using the O^p metric, with 42.55% of the program code to be examined by the programmer to locate the bug. From the figure, we also observe that the standard deviation of the respective programs is wider as the bug localization performance worsens.

We also evaluate on two sets of Space programs, using the entire test suite of Space, *AllTests* and using the subset of test suite of Space, *Subset* (in 10 bins). The details of

both sets of Space programs can be found in Section 4.5. Figure 5.13 shows the bug localization performance for the *AllTests* and *Subset* (in 10 bins) of the entire Space test suite, on the 15 single bug (one-bug) Space programs. From this figure, we observe the average rank percentages for *AllTests* are quite similar to the average rank percentages for the Space *Subset* (represented in 10 bins). We make a similar observation for the error bars of the *AllTests* and the *Subset* of all the 10 bins of the single bug Space programs.

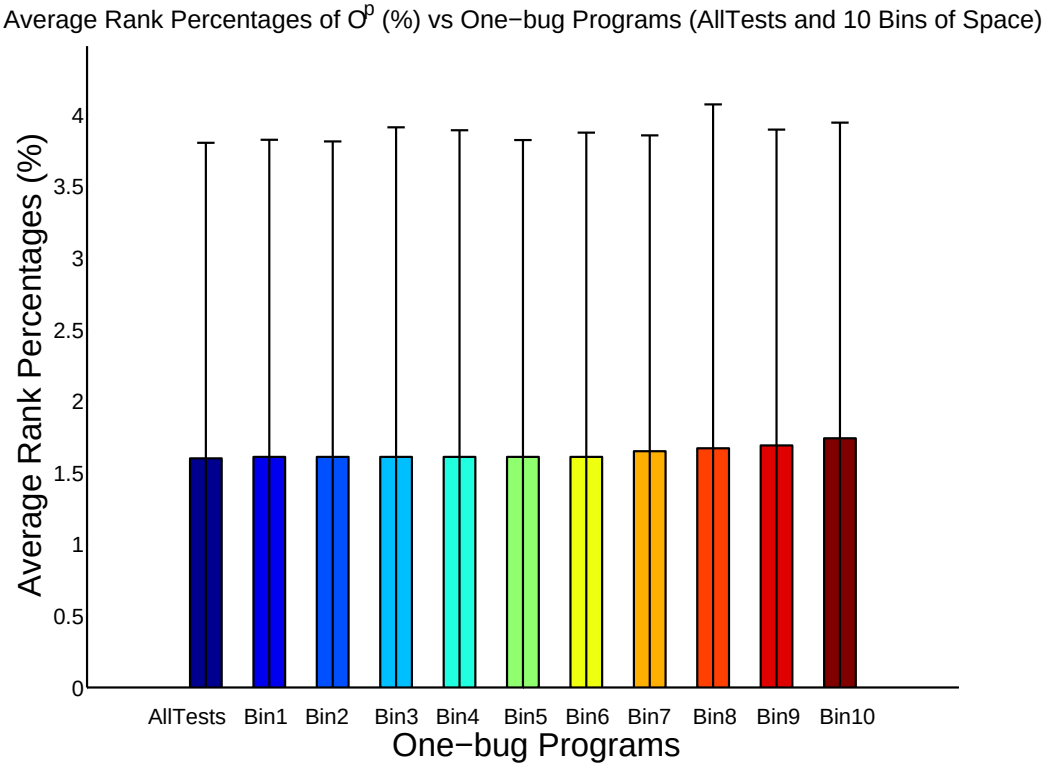

Figure 5.13: Average Rank Percentages with Error Bars evaluated using O^p metric on One-bug (Single Bug) Programs of Space

5.8.2 Successful Diagnosis of Bugs, $SucDiag$

We evaluate several spectra metrics using the performance measure of successful diagnosis of bugs, $SucDiag$. This refers to whether diagnosis is successful after examining some percentages of the program. We use this measure to perform a fair comparison of our proposed approach with other studies [Renieres and Reiss, 2003, Jones and Harrold, 2005, Cleve and Zeller, 2005, Hao et al., 2005, Jiang and Su, 2005, Liblit et al., 2005, Liu et al., 2005]. This measure has been described in detail in Subsection 4.3.2 in Chapter 4. We report the percentages of successful diagnosis of bugs over all the test suites of our benchmarks.

Table 5.13: Percentage of Successful Diagnosis of Bugs, $SucDiag$ for Single Bug Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space programs

Code Examined	1%	2%	4%	6%	8%	10%	20%	50%
O^p	11.42	22.69	37.55	45.93	51.12	54.81	67.60	91.34
O	11.42	22.69	37.55	45.93	51.12	54.81	67.60	91.34
Zoltar	11.02	22.69	37.41	45.59	50.81	54.06	67.03	90.88
Wong3	11.02	22.69	37.55	45.93	51.12	54.81	67.57	90.74
Wong4	10.82	21.84	36.09	44.43	49.48	53.02	66.38	89.70
M2	10.58	21.83	34.84	43.38	47.91	50.90	64.52	88.97
JacCube	10.53	21.81	34.85	43.62	48.28	51.24	64.79	88.95
McCon	10.21	21.48	36.34	45.09	49.79	52.77	65.56	90.16
Kulczynski2	10.21	21.48	36.34	45.09	49.79	52.77	65.56	90.16
Ample2	9.84	19.96	33.25	41.28	45.41	47.94	58.27	83.48
Ochiai	9.79	19.80	32.69	41.03	46.18	48.73	60.63	87.75
Ample	9.40	17.94	28.32	34.08	39.51	42.28	53.52	75.27
Rogot2	9.34	19.53	32.39	40.52	45.24	47.62	58.05	82.68
Pearson	9.31	19.30	32.16	40.09	45.60	47.90	58.88	84.22
AMean	9.11	18.34	30.94	39.21	44.31	47.13	58.68	84.24
Jaccard	8.89	17.37	29.87	36.37	41.73	45.19	58.32	85.52
CBI Log	8.36	14.78	26.15	32.90	38.49	41.17	53.64	81.76
Tarantula	7.64	14.42	25.24	31.61	37.24	40.63	53.38	80.82
Russell	1.43	4.04	8.84	13.40	18.08	22.03	40.46	82.93
Binary	1.43	4.04	8.84	13.40	18.08	22.03	40.46	82.93
Overlap	1.41	3.90	8.53	12.91	17.42	21.19	38.72	79.13

Table 5.13 shows the percentages of single bug programs for the Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space (combined benchmark sets) that are successfully diagnosed as the percentage of the code examined increases; lines of code that are not executed are ignored. Again, the results fit well with our model for the smaller percentiles. The performance measure used with our model, top-rank-bug score (Subsection 4.4.2), only considers the top-ranked statements — higher percentiles are ignored. Using average rank percentages (Subsection 4.3.1) arguably gives too much weight to high percentiles. A weighted average of rank percentages giving more weight to lower percentages would make a better practical measure of performance. We have experimented with such scoring functions with our model. The optimal metrics performed best in all cases examined. However, *proving* optimality for these more complex scoring functions is much more difficult than for the top-rank-bug score, evaluated on the model program in Section 5.7.

Table 5.14 gives the corresponding results for all the Siemens Test Suite programs with at least a fail test case (130 programs), which allows a fairer comparison with diagnosis

Table 5.14: Percentage of Successful Diagnosis of Bugs, $SucDiag$ for the Siemens Test Suite

Code Examined	1%	2%	4%	6%	8%	10%	20%	50%
O	11.29	22.19	39.58	50.05	55.60	59.34	70.42	89.57
O^p	11.29	22.19	39.58	50.05	55.60	59.34	70.42	89.57
Zoltar	11.29	22.19	39.39	49.67	55.60	59.34	70.42	89.57
Wong4	10.90	21.43	37.85	48.13	53.29	57.04	70.42	88.56
Wong3	10.52	22.19	39.58	50.05	55.60	59.34	70.42	87.65
Kulczynski2	9.75	20.66	37.47	48.67	53.88	57.63	69.34	89.49
McCon	9.75	20.66	37.47	48.67	53.88	57.63	69.34	89.49
M2	9.70	21.34	35.72	46.94	51.00	54.19	67.51	87.93
JacCube	9.66	21.29	35.48	46.43	50.70	53.90	67.27	87.61
Ochiai	9.03	19.04	32.59	43.30	47.25	50.04	61.43	86.69
Ample2	9.03	19.42	34.51	44.84	48.52	50.86	60.25	84.29
Rogot2	8.88	19.27	33.55	43.88	47.94	49.96	58.29	82.58
Pearson	8.88	18.89	33.17	43.11	47.90	50.54	60.06	84.32
Ample	8.65	17.67	27.34	33.22	39.15	42.01	52.50	72.36
AMean	8.50	17.22	31.05	42.34	46.15	48.48	59.97	84.98
CBI Log	8.34	15.59	28.57	37.61	41.51	45.50	57.65	83.82
Jaccard	8.11	15.89	29.24	37.57	42.90	46.88	59.52	84.33
Tarantula	7.96	15.58	28.55	37.57	41.45	45.42	57.51	83.56
Russell	1.62	4.15	8.87	13.73	18.82	23.09	41.82	82.40
Binary	1.62	4.15	8.87	13.73	18.82	23.09	41.82	82.40
Overlap	1.61	4.14	8.84	13.69	18.77	23.02	41.67	82.40
Jiang et al. [2005]	29.23	34.62	46.15	51.54	54.62	56.92	62.31	N/A
Sober	8.46	19.23	30.77	33.85	40.00	52.31	73.85	83.08
CBI	7.69	18.46	23.08	30.00	33.08	40.00	63.85	76.15
CT	4.65	N/A	N/A	N/A	N/A	26.36	37.98	60.47

based on other techniques (albeit with some noise from programs with no fail test cases). The table also lists the Wong3 and Wong4 metrics, which have been recently proposed in Wong et al. [2009]. The last line of the table gives figures for Cause Transition (CT) from Cleve et al. [2005]. Note that Cleve et al. [2005] only evaluates on 129 programs; therefore, we obtained their figures with respect to the 129 programs. The previous three lines of the table are figures published in Jiang et al. [2005] for their own system, SOBER [Liu et al., 2005], and CBI [Liblit et al., 2005]. Precise comparison is not possible because the way the percentiles of $SucDiag$ are computed is different: Jiang et al. [2005] use nodes in the program dependence graph, rather than lines of code that are executed. However, we can conclude that the system of Jiang et al. [2005] performs better than the best spectral-based diagnosis for the very small percentiles; it ranks significantly more bugs in the top 1%. This is unsurprising, as the analysis is much more sophisticated. However,

from the eighth percentile upwards, the best spectral methods (O, O^p, and Zoltar) appear to perform better.

The predicate-based CBI system performs worse than spectral-based diagnosis using the Tarantula metric. We show that the Tarantula metric is equivalent to our simplified version of the CBI metric (CBI Inc) in Proposition 5.2.6. The reason why the CBI metric performs worse than the Tarantula metric is the poor design of the CBI metric [Naish et al., 2010]. Recently, we investigated the relationship of statement-based spectra coverage and predicate-based spectra coverage (see the details of this study in Chapter 2) [Naish et al., 2010]. We reconstructed predicate-based spectra coverage and observed that the predicate-based CBI metrics perform more poorly than the simplified version of the CBI metric (CBI Inc). Our proposed FPC metric (Failure *plus* Context) has been shown to perform better than the predicate-based CBI metric (CBI Inc). These metrics can be found in Table 2.5. The other possible explanation of the poor performance of the predicate-based CBI metric is that the way the figures are calculated makes the comparison misleading (for example, basic blocks versus statements or the treatment of ties in the ranking).

5.9 Multiple-bug Programs

We evaluate the same set of spectra metrics on the multiple-bug programs (the two-bug and three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite). We combine all the single bug programs in order to generate the multiple-bug programs and the details can be found in Section 4.5. We also evaluate the multiple-bug Space programs, which consist of two-bug, three-bug, and more than three-bug (see Table 4.1).

Table 5.15 shows the breakdown and overall performance of bug localization for the two-bug programs Siemens Test Suite and the subset of the Unix Test Suite using several spectra metrics. As expected, the O and O^p metrics, which are optimal for single bug programs, are no longer the best performing metrics for the two-bug programs. We observe a different ordering of metrics in this table. Kulczynski2 metric yields the best bug localization performance overall (Combined column). Using this metric, the programmer only needs to examine 19.53% of the program code in order to locate the two-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite. The Kulczynski2 and McCon, and Pearson metrics give the least average rank percentages for the Siemens Test Suite and the subset of the Unix Test Suite, with 18.25% and 22.06%, respectively. In the Combined column of this table, we observe O^p shows better bug localization performance as compared to the O metric, with 22.84% and 24.95%, respectively. A buggy statement is not necessarily executed by all the fail test cases in a multiple-bug program. In the O metric, statements with a_{ef} less than $totF$ are given a minimum metric value of -1. The O^p

Table 5.15: Average Rank Percentages for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite — executed lines of code only

Benchmark	Siemens	Unix	Combined
Kulczynski2	18.25	22.74	19.53
McCon	18.25	22.74	19.53
Ochiai	19.20	22.60	20.18
JacCube	18.67	25.10	20.51
Zoltar	19.11	24.06	20.52
Wong4	20.21	22.19	20.78
M2	19.01	25.26	20.80
AMean	20.47	22.23	20.97
Pearson	20.70	22.06	21.09
Jaccard	20.49	22.64	21.10
Ample2	20.72	23.00	21.37
Rogot2	21.62	22.18	21.78
CBI Log	21.65	23.81	22.26
Wong3	20.89	26.67	22.54
O^p	21.13	27.10	22.84
Tarantula	21.66	26.82	23.13
Ample	24.77	23.80	24.49
O	22.66	30.67	24.95
Russell	31.80	32.84	32.10
Binary	33.13	36.26	34.02
Overlap	31.18	41.09	34.02

metric ranks primarily on a_{ef} followed by a_{ep} (see Subsection 5.5.2). Therefore, the buggy statements in the multiple-bug programs can be ranked higher using O^p as compared to the O metric.

Figure 5.14 shows the plot of the average rank percentages for better performing metric, Kulczynski2, on the two-bug programs in the Siemens Test Suite and the subset of the Unix Test Suite. The tot_info (*Tot*) program shows the best bug localization performance for the two-bug programs, with the programmer only examining 7.37% of the program code in order to locate bugs. On the other hand, the Col program shows the worst bug localization performance for the two-bug programs. The programmer has to examine 28.07% of the program code in order to locate bugs. The error bars in this figure are widest for the Tr, Sch2, and Col programs, as they show the worst bug localization performance in the two-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite.

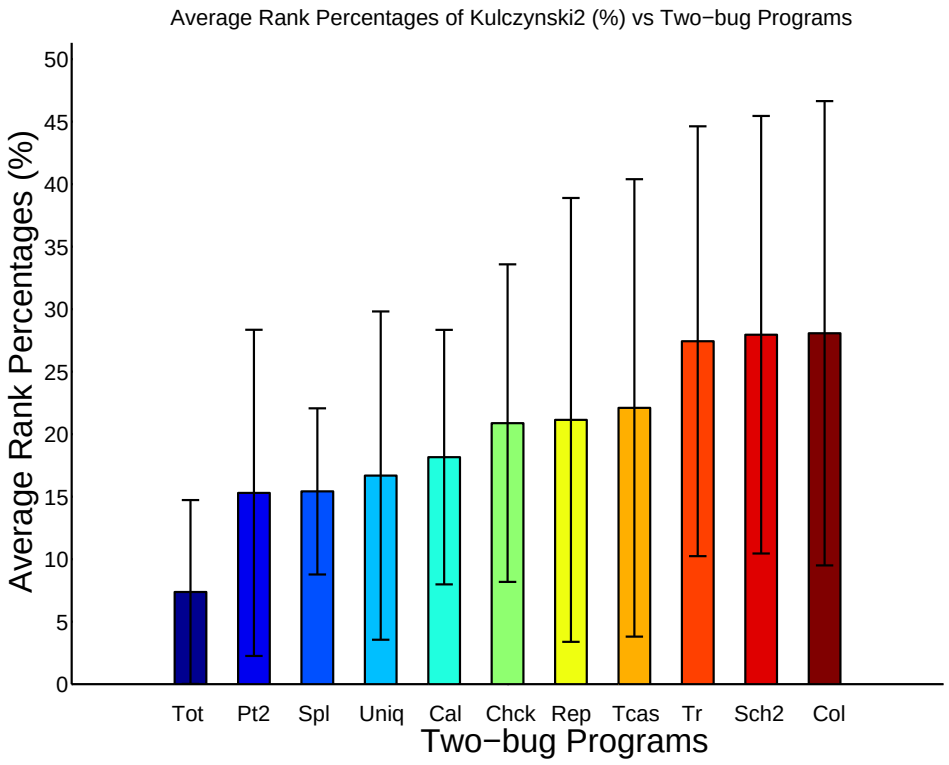

Figure 5.14: Average Rank Percentages with Error Bars (Standard Deviation) evaluated using Kulczynski2 metric on the Two-bug Programs of Siemens Test Suite and the subset of the Unix Test Suite

Table 5.16 shows the performance of bug localization for the three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite, evaluated with several spectra metrics. In this table, the overall ordering of the metrics is different from the ordering of the metrics for the two-bug programs (Table 5.15). Again, we observe that the O^p metric is not the best spectra metric for the three-bug programs. The Kulczynski2 and McCon metrics have the best bug localization performance (least average rank percentages) among all the other metrics for three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite. By using the latter metrics, the programmer only needs to examine 21.94% of the program code in order to locate the bugs.

Figure 5.15 shows the average rank percentages for the Kulczynski2 metric evaluated on the three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite. The tot_info (*Tot*) program shows the best bug localization performance among the three-bug programs. The programmer has to examine only 5.84% of the program code in order to locate bugs. The error bar in this figure is widest for the print_tokens2 (*Pt2*) program.

Table 5.16: Average Rank Percentages for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite — executed lines of code only

Benchmark	Siemens	Unix	Combined
Kulczynski2	8.70	24.88	21.94
McCon	8.70	24.88	21.94
AMean	15.42	23.75	22.24
Pearson	15.73	23.77	22.31
Rogot2	16.94	23.78	22.53
Ample2	16.24	23.96	22.56
Ochiai	12.59	24.82	22.60
Wong4	12.04	24.94	22.60
Jaccard	15.32	24.85	23.12
CBI Log	18.08	24.54	23.37
Zoltar	8.59	26.80	23.49
Ample	20.11	24.72	23.88
JacCube	14.19	27.23	24.86
M2	14.74	27.24	24.97
Wong3	12.45	28.08	25.24
O^p	12.51	28.18	25.33
Tarantula	18.17	29.24	27.23
Russell	18.22	30.99	28.67
O	14.54	32.56	29.29
Binary	19.64	35.27	32.43
Overlap	16.18	43.26	38.34

Table 5.17 shows the bug localization performance of multiple-bug Space programs. As described in Section 4.5, we use two sets of Space programs. The first set of Space programs that we evaluate is using the entire Space test suite, which is known as *AllTests*. The other set of Space programs that we evaluate is the subset of the entire test suite of Space in 10 bins (*Subset* column). The ordering of the metrics for both sets are very similar, except for the Wong4 and Jaccard metrics.

Figure 5.16 shows the plot of the bug localization performance on *AllTests* and the *Subset* of the entire Space test suite (in 10 bins) of 13 multiple-bug Space programs. From the figure, we observe the average rank percentages using the top performing metric, Zoltar for the *AllTests* is quite similar to the average rank percentages of Zoltar for the Space *Subset* (represented in 10 bins). We make the same observation on the error bars (standard deviation) of the *AllTests* and the *Subset* of all the 10 bins of the multiple-bug Space programs.

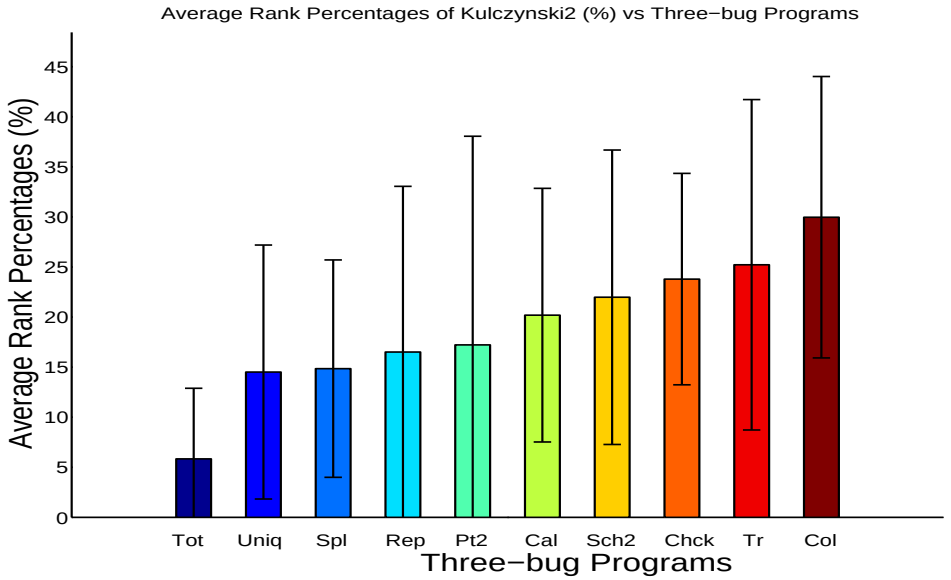

Figure 5.15: Average Rank Percentages with Error Bars (Standard Deviation) evaluated using Kulczynski2 metric on the Three-bug Programs of Siemens Test Suite and the subset of the Unix Test Suite

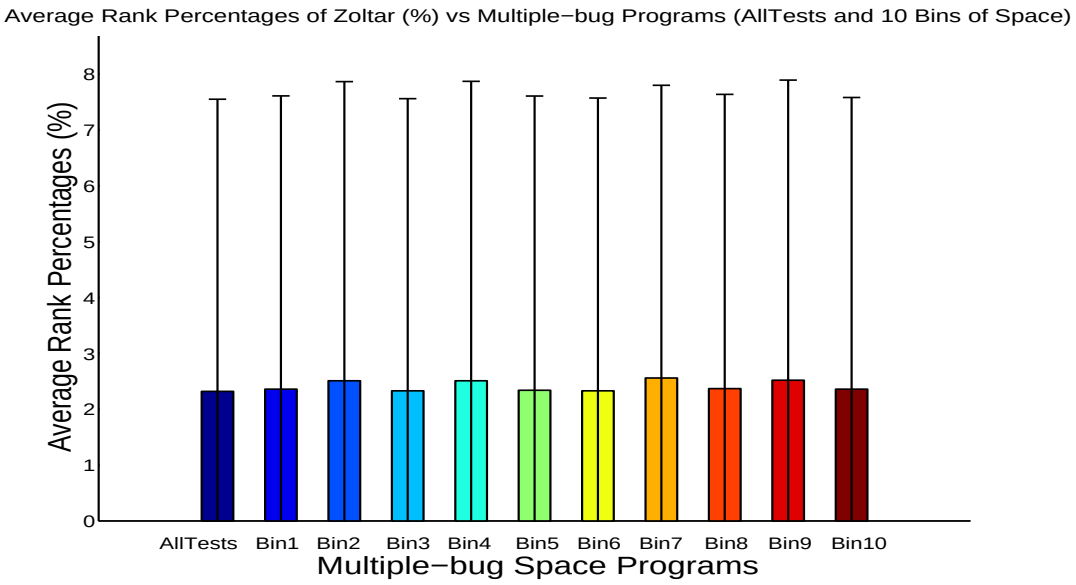

Figure 5.16: Average Rank Percentages with Error Bars (Standard Deviation) evaluated using Zoltar metric on the Multiple-bug Space Programs

Table 5.17: Average Rank Percentages for the AllTests and *Subset* of Multiple-bug Space Programs (average of 10 bins)

Metric	Space	
	AllTests	Subset
Zoltar	2.32	2.42
JacCube	2.42	2.49
O	2.50	2.54
O^p	2.50	2.54
Wong3	2.50	2.54
Wong4	2.50	2.82
M2	2.58	2.58
Kulczynski2	2.61	2.64
McCon	2.61	2.64
Ochiai	2.65	2.85
Rogot2	3.22	3.42
Pearson	3.22	3.49
Jaccard	3.25	3.37
Ample2	3.39	3.52
AMean	3.47	3.65
CBI Log	4.41	3.85
Tarantula	4.51	4.44
Ample	8.13	8.09
Overlap	17.50	18.07
Russell	17.50	17.85
Binary	17.50	17.85

5.10 Discussion

We have shown that parameters such as error detection accuracy, q_e, proportion of fail tests, the number of fail tests, and buggy code execution frequency, can be explicitly adjusted (Subsection 5.7.2, 5.7.3, and 5.7.4 respectively) in the model program to observe their relationships with the bug localization performance. More investigation could be done to fine tune these parameters of the model program. The model program that we proposed does not have to be restricted to the two if-then-else statements; rather, it can be defined for a nested if-then-else or a while loop.

We have explored using different models of multiple-bug programs tuned with the above mentioned parameters. For models with multiple bugs, we see a divergence between the performance of O and O^p. O^p and the other good metrics other than O generally perform quite well, especially when most failures are caused by a single bug. In some cases, Ochiai and even Jaccard perform better. None of the metrics examined is optimal

for more than a very small number of tests. Using different model programs, we have constructed optimal metrics for some cases. However, we have not successfully obtained similar orderings of the metrics using the models and the multiple-bug program benchmarks (Table 5.15, Table 5.16, and Table 5.17). It may be that by constructing optimal metrics in other cases, some patterns will emerge. This would allow us to find optimal or near-optimal metrics in a broad range of cases. It may be also due to the multiple-bug programs benchmarks which we have used. These multiple-bug programs are generated by the combination of single bug program versions (details can be found in Section 4.5). Some of these programs have more program versions as compared to the other programs. For example in Table 4.1, we observe the $tcas$ program has 604 two-bug programs as compared to only 28 two-bug programs versions for $schedule2$ (Sch2). With this imbalanced distribution of the number of programs, it is possible that we cannot obtain an accurate representation of the bug localization performance for the multiple-bug programs by using the spectra metrics.

5.11 Other Performance Measures

Several performance measures have been proposed in Chapter 4. We evaluate most of these measures on single bug programs of the Siemens Test Suite benchmark. The evaluation results on multiple-bug program benchmarks are not considered in this section, as the optimality of spectra metric for multiple-bug programs has not been generalised. However, we have evaluated O^p metric in Section 5.9 to see how well this metric performs for multiple-bug programs.

5.11.1 High, Mid, Low, and Median measures

In Section 4.2, different measures are detailed to handle ties between statements having the same metric value as the buggy statement. These measures have been used in previous studies [Jones and Harrold, 2005, Abreu et al., 2006, Abreu et al., 2007, Wong et al., 2010]. For fair comparison, we evaluate the *High*, *Mid*, and *Low* (Section 4.2) of the rank percentages measure in all the single bug programs of the Siemens Test Suite. For these measures, we consider the average of the rank percentages of these programs.

Table 5.18 shows the different measures of the average rank percentages for several metrics in single bug programs of the Siemens Test Suite, with the optimal metrics, O and O^p, performing the best. By using the latter metrics, the programmer needs to examine 10.69%, 15.69%, and 20.68% of the program code in order to locate the bug using the *High*, *Mid*, and *Low* measures respectively. The *Mid* measure refers to the average rank

Table 5.18: Results of High, Mid, Low, Median, First Quartile, and Third Quartile Rank Percentages for the Siemens Test Suite (Single Bug Programs)

Metric	High	Mid	Low	Median	First Quartile	Third Quartile
O, O^p	10.69	15.69	20.68	5.80	2.34	24.57
Zoltar	10.72	15.71	20.71	6.10	2.34	24.57
Kulczynski2	11.13	16.13	21.12	6.25	2.34	24.57
McCon	11.13	16.13	21.12	6.25	2.34	24.57
Wong4	11.27	16.27	21.28	6.85	2.34	25.78
Wong3	11.29	16.28	21.28	5.80	2.34	25.78
M2	13.29	18.28	23.28	7.63	2.44	27.34
JacCube	13.32	18.31	23.31	8.05	2.54	27.34
Ochiai	15.17	20.17	25.16	9.95	2.76	27.34
AMean	16.82	21.81	26.80	12.25	2.80	30.77
Pearson	17.08	22.07	27.06	12.43	2.76	33.34
Ample2	17.21	22.21	27.20	11.61	2.54	33.47
Jaccard	18.48	23.47	28.47	17.48	2.80	35.89
Rogot2	18.84	23.84	28.83	15.47	2.54	37.29
Tarantula	19.13	24.12	29.12	18.62	2.80	38.69
CBI Log	18.79	24.46	30.13	19.09	2.80	40.57
Russell	0.90	28.36	55.82	29.87	21.77	37.71
Binary	0.90	28.36	55.82	29.87	21.77	37.71
Overlap	0.90	28.39	55.88	29.87	21.77	37.71
Ample	25.82	31.39	36.96	18.30	3.04	67.97

percentages measure that we use to report bug localization performance figures throughout the thesis. Therefore, the figures for the *Mid* measure are the average rank percentages figures we reported in the *Siemens* column of Table 5.11. For the *High* measure, Russell, Binary, and Overlap are the metrics that give the smallest average rank percentages (0.90%) as compared to other metrics. This does not indicate that the latter are the best metrics. Rather, this observation is due to the nature of these metrics. The statement initialisation codes are always executed by all the test cases. We know that the buggy statement for single bug programs is executed by all the fail test cases. Therefore, by using the Russell and Binary metrics (see the metrics in Table 2.3), the statement initialisation codes will yield identical metric values and rank equally with the buggy statement. Using the *High* measure for the Russell and Binary metrics return the first ranking position of the statement that shares the same metric value as the buggy statement. For the Overlap metric, the denominator of the metric always takes the minimum value among a_{ef}, a_{nf}, and a_{ep} (see the metric in Table 2.3). The initialisation code always has the same value for both numerator and denominator of the buggy statement. The same case applies for Russell, Binary, and Overlap metrics when evaluating using the *Low* measure. Due

to ties in the ranking, these metrics return the last ranking position of the statement that shares the same metric value as the buggy statement. Therefore, it is rather misleading to use extreme cases, such as the *High* and *Low* measures.

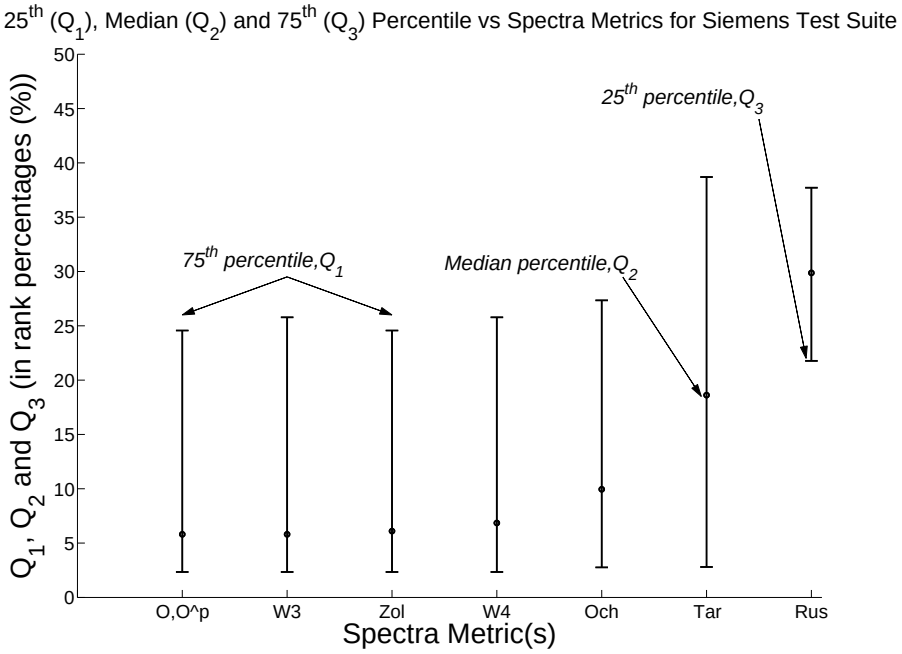

Figure 5.17: Median Rank Percentages of the Siemens Test Suite (Single Bug) Programs

Instead of taking the average of the rank percentages using the *High*, *Mid*, and *Low* measures, we consider the median, First Quartile, and Third Quartile [Dodd, 1938]. The median rank percentages is used to avoid any outlier in bug localization performance across the datasets. By using the average rank percentages, there is a possibility that several programs that perform very poorly in the bug localization performance would affect and distort the overall bug localization performance of a particular spectra metric. The First Quartile (25th Percentile) and Third Quartile (75th Percentile) are used to understand the spread of the percentages of programs in the test suite that perform in the respective average rank percentages on several spectra metrics. We consider the *Mid* measure to evaluate the median, First Quartile, and Third Quartile measures.

We also observe that the First Quartile and Third Quartile of the rank percentages for several metrics at the top of the table are identical. We observe that the spread of the rank percentages (difference between the 25th and 75th percentile) gets larger for metrics that are ranked at the bottom of the table. This indicates that by using these metrics, buggy statements in the single bug programs do not rank high most of the time. We plot the median rank percentages and the error bar (representing the First Quartile (Q1) and Third Quartile (Q3)) for several metrics such as O, O^p, Wong3 (W3), Zoltar (Zol), Wong4 (W4), Ochiai (Och), Tarantula (Tar), and Russell (Rus), in Figure 5.17. We observe that

the Russell (Rus) metric has the narrowest error bar in the figure. It ranges from the rank percentages of approximately 21% to 38%. By using the metric, there are huge number of programs where the non-buggy statements ranked equally with the buggy statement. Therefore, the rank percentages for the single bug programs evaluated with the Russell (Rus) metric do not differ much.

5.11.2 Top-rank-bug Score

We have evaluated the top-rank-bug score on our $ITE2_8$ model program in Table 5.1. In Table 5.19, we detail the breakdown evaluation of top-rank-bug score on all the single bug programs of the Siemens Test Suite. We observe that the O and O^p metrics are among the top metrics, with 10.94% of the single bug programs of the Siemens Test Suite having the bug ranked top.

Table 5.19: Results of the Top-rank-bug Score (%) for the Siemens Test Suite (Single Bug Programs)

Metric	Top-rank-bug (%)
O,O^p	10.94
Zoltar	10.94
Wong4	10.94
Wong3	10.12
Kulczynski2	9.713
McCon	9.713
M2	9.664
JacCube	9.246
Ample2	9.25
Ample	9.217
Rogot2	9.082
Ochiai	8.836
AMean	8.672
Pearson	8.672
Jaccard	8.262
CBI Log	8.111
Tarantula	8.098
Russell	1.949
Binary	1.949
Overlap	1.943

5.11.3 Relative Score

In Subsection 4.4.3 of Chapter 4, we have defined the *relative score* measure Rel_p (see Definition 13). This measure is used to evaluate bug localization performance of a spectra metric in the cases where no sensible metric is able to rank the bug on top. One of the observed cases is a huge number of non-buggy statements sharing identical a_{ij} values with the buggy statement in the subset of the Unix Test Suite in Subsection 6.3.2. It causes ties of metric value of the non-buggy statements with the buggy statement. Due to the ties in the ranking of the statements, no sensible spectra metric can rank the bug of the program on top and affects the bug localization performance. This measure ignores such case and evaluate the best possible bug localization performance a spectra metric could achieve.

Table 5.20: Results of Average Relative Score for Single Bug Programs of the Siemens Test Suite, the subset of Unix Test Suite, Concordance, and Space

Metric	Average of Relative Score
O,O^p	100.00
Zoltar	99.65
Wong3	99.70
Wong4	98.97
Kulczynski2	98.88
McCon	98.88
JacCube	97.98
M2	97.89
Ochiai	96.51
Pearson	94.40
Jaccard	94.35
AMean	94.28
Ample2	93.95
Rogot2	93.18
CBI Log	90.96
Tarantula	90.94
Ample	87.82
Russell	83.67
Binary	83.67
Overlap	81.62

We evaluate the relative score measure on our single bug and multiple-bug benchmarks, namely the Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space. For the Space, we evaluate the measure on the entire test suite of the Space, *AllTests*. We report the average of the relative score with respect to the single bug and multiple-bug benchmarks. Table 5.20 shows the average relative score for several spectra

metrics evaluated on the 250 single bug programs of our benchmarks. Higher relative score of a metric (see Definition 13) indicates the best bug localization performance that a metric can achieve. The table shows that O and O^p metrics have the highest average relative score, 100 compared to other metrics.

In the design of our optimal metrics O and O^p for single bug programs (see Section 5.5), the buggy statement must be executed by all the fail test cases ($a_{ef} = totF$). O metric returns a_{np} if it is a buggy statement. O^p metric primarily ranks statements based on the a_{ef} and then a_{ep}. As explained in Subsection 4.4.3, using any sensible metrics, statement(s) that rank(s) higher than the buggy statement have a_{ef} and a_{np} values greater than the buggy statement's a_{ef} and a_{np} respectively. For the latter case, no sensible metric can achieve any better bug localization performance and rank the bug top in the ranking. For O and O^p metrics in single bug programs, we know that any statement that ranks higher than the buggy statement, must have $a_{ef} = totF$ and a_{np} values greater than the buggy statement's a_{np} value. Using relative score (Algorithm 29), statements that ranked higher than the buggy statement with the latter conditions are not considered. Relative score also ignore the case where the statement(s) that rank(s) higher than the buggy statement that has the identical a_{ij} values (having similar metric value) as the buggy statement. Therefore, *nonbug* and *nonbugties* variables of Algorithm 29 are 0 for O and O^p metrics. These metrics are able to show the best bug localization performance with the relative score of 100 for all the single bug programs.

We observe other metric such as Tarantula has the average relative score of 90.94. The reason to the different average relative score in O^p and Tarantula metrics is due to the influence of a_{ep} in the denominator of Tarantula metric. Some of the non-buggy statements have lesser a_{ep} values than the buggy statement's a_{ep} values. Even though these non-buggy statements have lesser a_{ef} value than the buggy statement, Tarantula metric can possibly rank these non-buggy statements higher than the buggy statement. Therefore, Tarantula metric is unable to show the best bug localization performance as compared to using the O^p metric on programs with the latter condition.

We also evaluate the average of the relative score for 6661 multiple-bug programs of our benchmarks (Siemens Test Suite, subset of the Unix Test Suite, and Space) in Table 5.21. Kulczynski2 and McCon metrics yield the highest average relative score of 98.09 as compared to other metrics. Unlike in Table 5.20 where better performing metrics, O and O^p metrics show the best possible bug localization performance (100), better performing metrics such as, Kulczynski2 and McCon metrics in Table 5.21 do not yield average relative score of 100 for the multiple-bug programs of our benchmarks. The latter observation might be due to the multiple-bug program benchmarks that we use. This has been discussed in detail in Section 5.10. It may also be other reasonable spectra

metrics that are possible to achieve better bug localization performance than Kulczynski2 and McCon metrics on the multiple-bug program benchmarks and these will be the future work.

In all the evaluation on single bug programs of the Siemens Test Suite using different performance measures in this section, the ordering for most of the better performing spectra metrics are similar to the ordering of the metrics using the average rank percentages (Siemens column of Table 5.11). We observe that bug localization performance using optimal metrics O and O^p on the Siemens Test Suite single bug programs consistently outperform other metrics across the different performance measures. Wong3 and Zoltar metrics show some slight different orderings on several performance measures, but the differences between these metrics are small.

Table 5.21: Results of Average Relative Score for Multiple-bug Programs of the Siemens Test Suite, the subset of Unix Test Suite, and Space

Metric	Average of Relative Score
Kulczynski2	98.09
McCon	98.09
Wong4	97.95
Zoltar	97.83
Ochiai	97.06
AMean	96.61
Pearson	96.53
Ample2	96.42
Jaccard	96.31
Rogot2	96.19
JacCube	96.06
Wong3	95.94
M2	95.92
O^p	95.91
CBI Log	95.72
Ample	94.53
O	94.37
Tarantula	93.51
Russell	91.85
Binary	90.43
Overlap	88.46

5.12 Summary

In this chapter, we have advanced the state-of-the-art of software fault diagnosis using program spectra. We proposed a simple *model* to evaluate the performance of spectra metrics. We performed extensive empirical study of metrics that can be used for ranking statements according to how likely they are buggy. We also showed several metrics are equivalent for ranking using the property of monotonically increasing function. Based on the *model*, we developed a theoretical understanding of single bug programs that led the discovery of optimal spectra metrics for single bug programs that are much simpler than many of the metrics proposed. The evaluation results from the model fit very well with the results for single bug programs in the benchmarks (Siemens test suite, subset of the Unix Test Suite, Concordance, and Space), where the optimal metrics perform the best. We considered this to be a firm validation of our approach. Multiple-bug programs were also evaluated using the same set of metrics. This serves as the motivation to propose several bug localization approaches and to improve the bug localization performance for multiple-bug programs. We briefly discussed using the proposed model program for multiple-bug programs. Finally, several new performance measures were evaluated especially on the single bug Siemens Test Suite to gain insights of these measures using several spectra metrics.

6

Bug Consistency of Buggy Statement with respect to Bug Localization Performance

6.1 Introduction

There are two types of buggy statement in a typical buggy program, namely deterministic and non-deterministic buggy statements [Liblit et al., 2003]. Deterministic buggy statements are statements that always show unintended output when executed by any test case; for example, when any of the test cases executes the buggy statement, variable *x* returns the wrong value instead of the expected value. Non-deterministic buggy statements are statements that only show unintended output some time when executed by any of the test cases. This type of bug is usually harder to diagnose, as the buggy statement does not always show unintended output when executed by any of the test cases. These two classifications essentially refer to bug consistency (how consistently the buggy statement shows unintended output when executed by test cases). We are interested in investigating the relationship between the consistency of the bug and bug localization performance.

Abreu et al. have introduced a measure of error detection accuracy, q_e [Abreu et al., 2007]. We discuss how the effectiveness in determining buggy statements (bug localization performance using rank percentages) for various metrics degrades as the bug consistency (error detection accuracy, q_e) of a statement approaches zero. We show the relationship between bug localization performance using rank percentages and bug consistency, q_e using several spectra metrics on single bug and multiple-bug programs.

6.2 Relationship of Bug Consistency, q_e with respect to Bug Localization Performance (Rank Percentages)

The *bug consistency* of a buggy statement is defined as how frequently it shows unintended output when executed. *Bug consistency* fits into the definition of error detection accuracy, q_e, which has been used as a measure of the accuracy with which the bug is detected in a program [Abreu et al., 2007]. q_e is defined as the proportion of fail tests in test cases where the buggy statement was executed. We have already defined q_e (Definition 18) in page 113. In this chapter, we use the terms *bug consistency* and q_e interchangeably.

We consider a good approach to locate bugs should have maximum bug localization performance for consistent (deterministic) bugs. As bug consistency approaches zero, bug detection becomes harder, and the prediction becomes no better than a random guess. In the worst case, the bugs will be so inconsistent that there can be no failures in the test suite used. The bug consistency, q_e, value depends on the particular test suite used.

We show the relationship of bug consistency with bug localization performance using the rank percentages performance measure (Subsection 4.3.1 of Chapter 4) with the following hypothesis. Smaller rank percentages means less program code needs to be examined by the programmer in order to locate a bug. As q_e approaches 0, the ranking of the buggy statement can vary between lower and higher ranks. As q_e approaches 1, the buggy statement will be ranked higher, approaching the best possible bug localization performance (smallest rank percentages).

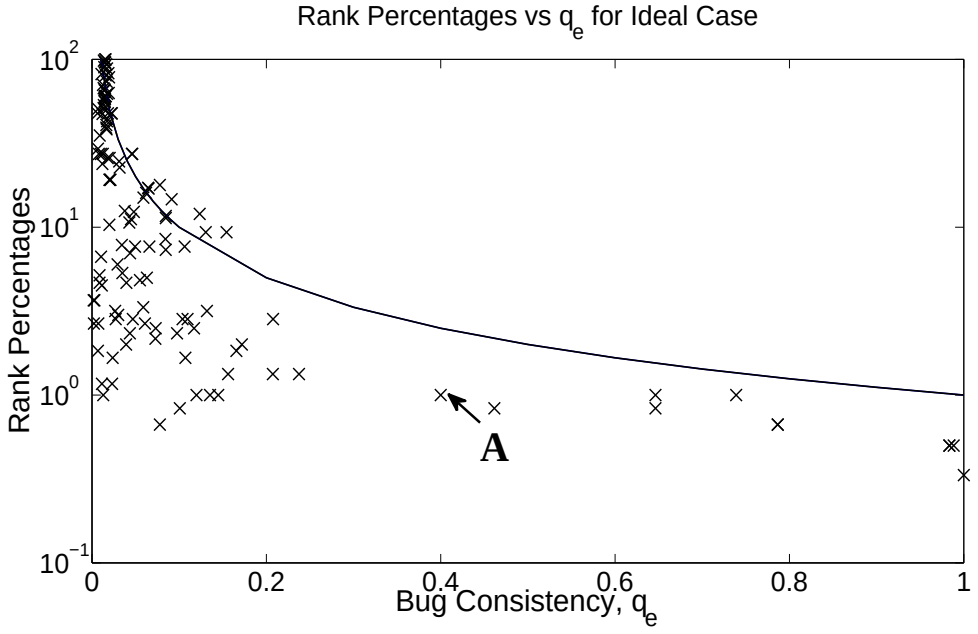

Figure 6.1: Relationship of Rank Percentages vs q_e for *Ideal Case*

Based on the above hypothesis, we plot the *Ideal Case* in Figure 6.1. Each point in this figure represents a program which has a buggy statement. In this figure, we make an assumption that a typical program has 300 program statements. The figure shows bug localization performance – how much program code must be examined in order to locate the buggy statement – as the q_e value of the buggy statement varies. We use a logarithmic scale for the y-axis to observe programs with low rank percentages. Point **A** in the figure refers to a typical program where the buggy statement has a q_e value of 0.4 and requires the programmer to examine 1% of the program code (3 lines of code) to locate the buggy statement.

In the figure, as the q_e is approaching 1 (the bug is consistent), the bug localization performance converges to lower rank percentages. The buggy statement that ranks first among the 300 program statements gives us a rank percentage of approximately 0.33%. This is the best possible bug localization performance (using rank percentages). The points in the figure form a asymptotically decreasing curve for the *Ideal Case*. For better performing spectra metrics on single bug programs (observed in Chapter 5), such as O and O^p, we expect the bug localization performance to improve as the bug becomes consistent (q_e is 1). Therefore, we expect the points for these metrics to form a similar asymptotically decreasing curve of the *Ideal Case*.

For metrics that perform less well than the O and O^p metrics, such as Tarantula and Rogers, as the q_e increases (the bug becomes more consistent), the buggy statement will not be located as quickly as for the better performing spectra metrics. Therefore, the points for these metrics would form a less steep asymptotically decreasing curve, as bug localization performance improves much slower than the better performing spectra metrics.

6.3 Plot of the Relationship of Rank Percentages vs Bug Consistency, q_e

In this section, we plot the relationship between rank percentages and bug consistency, q_e, for several spectra metrics. Abreu et al. performed a similar study of the relationship between rank percentages and the q_e [Abreu et al., 2007]. However, they modify the test cases in the test suite of the benchmark in order to vary the q_e values. They randomly exclude pass and fail test cases for the programs in the Siemens Test Suite [Do et al., 2005] in order to achieve this. In our study, we use the entire Siemens Test Suite to perform the study of this relationship. We also use other benchmarks such as the subset of the Unix Test Suite, Space, and Concordance to study the same relationship. We plot

the relationship for different benchmarks of single bug programs and compare the latter relationships with the *Ideal Case* of Figure 6.1. Both O and O^p metrics are optimal metrics for single bug programs in Chapter 5. Therefore, we only show the plot for one of the optimal metrics, O^p, in this chapter.

6.3.1 Siemens Test Suite

Initially, we plot rank percentages and bug consistency, q_e for the single bug programs of the Siemens Test Suite programs for the two spectra metrics, namely O^p and Rogers (Figure 6.2 and Figure 6.3). We have observed that O^p metric is a better performing metric for the Siemens Test Suite in Table 5.11 in Chapter 5. Rogers metric does not perform so well and is part of the *Group B* metrics in Table 5.12.

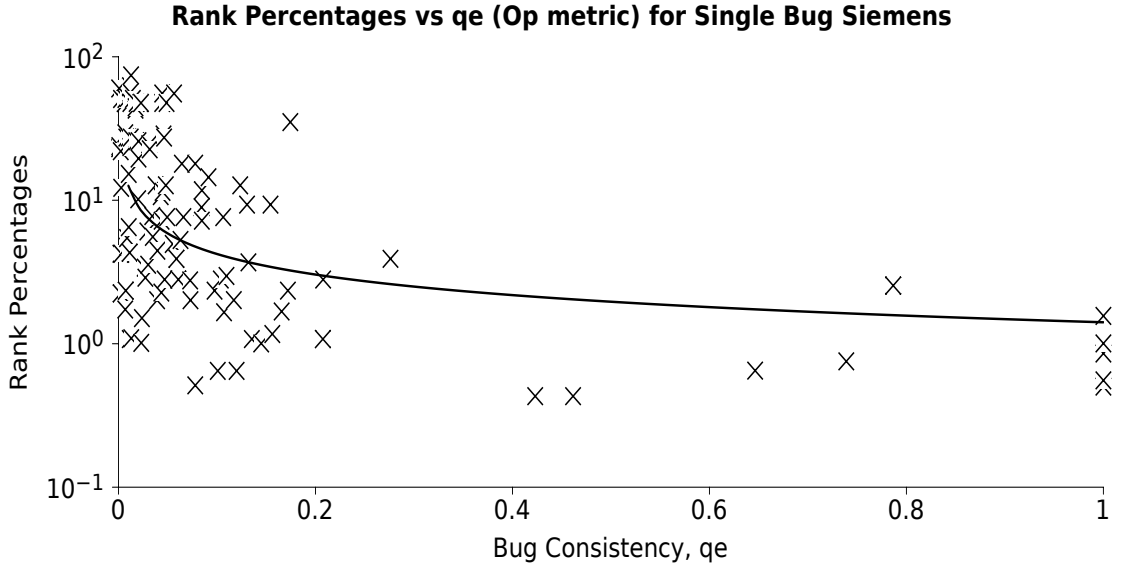

Figure 6.2: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the O^p metric

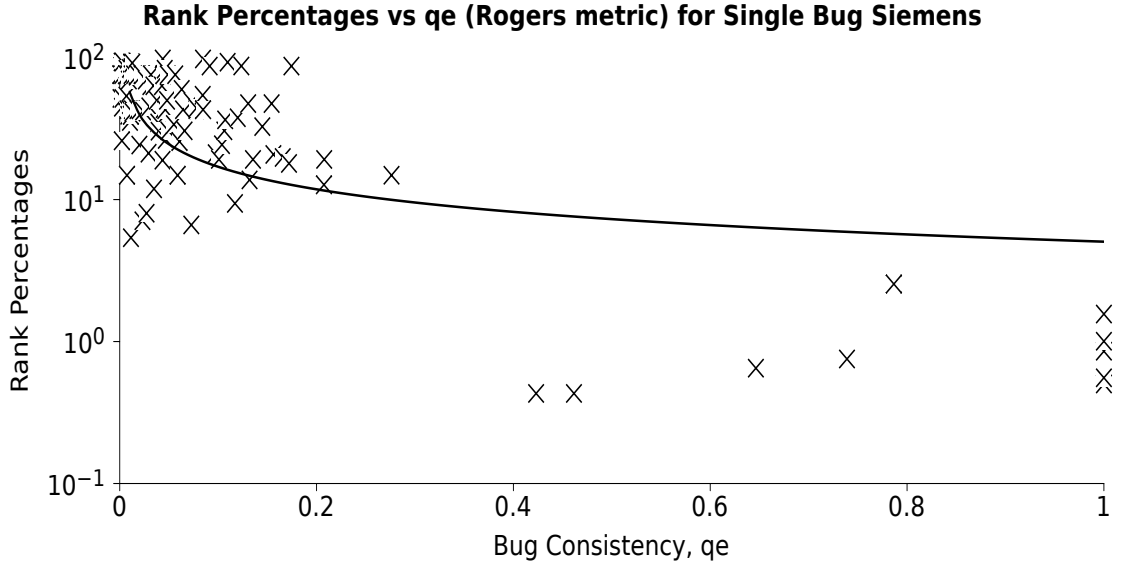

Figure 6.3: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the Rogers metric

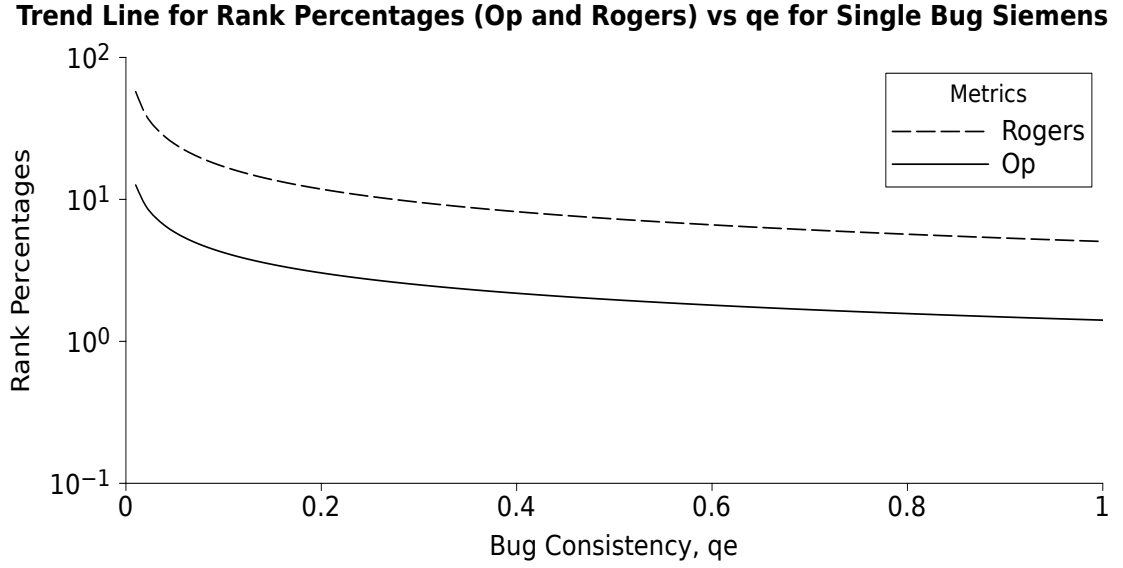

Figure 6.4: Trend line for the Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the O^p and Rogers metrics

Figure 6.2 and Figure 6.3 show the rank percentages for the O^p and Rogers metrics, compared to bug consistency, q_e. Each point, represented with the symbol of X in the figure, relates to the buggy statement of a typical program of 122 single bug Siemens Test Suite programs. For illustrative purposes, we plot the trend lines based on the points in both of these figures using a fitting function, known as the power regression function [Weisstein, 2011]. This function is also known as least square fitting function with power law, and is commonly used to study the trend of data in several biochemical stud-

ies [Savageau, 1969, Shiraishi and Savageau, 1992]. We choose to use this fitting function as it is readily available in the Gnumeric tool [GNOME, 2010]. Other fitting functions could also be used for the plot of the trend lines in this study.

For both of the figures, we observe that there are many points along the y-axis when the q_e value is small. These points indicate that the ranking of the buggy statement is anywhere between the lower and the higher rank position. In the Rogers metric (Figure 6.3), we observe most of the points concentrate between the rank percentages of 10% to 100% when the q_e value is no more than 0.3. As the q_e value increases and approaches 1, we observe the rank percentages becoming smaller for both of the metrics. This indicates that the buggy statement is ranked at a higher position as the bug becomes more consistent. The programmer can easily locate the bug without needing to examine much program code. This supports the hypothesis we established earlier in Section 6.2.

When the q_e value is 1, we also observe that some of the points are still in the range of rank percentages of 0.6% to 2% in both Figure 6.2 and Figure 6.3. These points are due to several programs in the Siemens Test Suite having ties, that is, buggy statement that share the same a_{ij} values and metric value as the non-buggy statements, where these non-buggy statements are in the same block of the program as the buggy statement. Such a case is more obvious in the subset of the Unix Test Suite, and we defer the discussion to Subsection 6.3.3.

For clarity purposes, we separately show the trend lines observed in Figure 6.2 and Figure 6.3 in Figure 6.4. We observe the difference in the trend lines for the O^p and Rogers metrics. The latter metric shows a less steep asymptotic decreasing curve as compared to the former. This indicates that as q_e increases, the bug localization performance of the Rogers metric does not improve as much as compared to a better performing metric such as the O^p metric.

The Russell metric in Figure 6.5 provides a different observation. The trend line for the metric shows a flat horizontal line for all the q_e values. In this figure, most of the points are in the rank percentages of 30%-60% for all the q_e values. The reason for this observation is the nature of the metric (see the metric in Table 2.3). The Russell metric ranks the statements of the program based on the a_{ef} value. The buggy statement is ranked together with the other non-buggy statements (e.g statement initialisation), producing a huge number of ties. In this thesis, the *Mid* measure (for details of this measure, refer to Section 4.2) is used to handle the ranking of a buggy statement that has ties with non-buggy statements. Therefore, the rank percentages obtained using this metric reflect the average of the bug position in the ties, and do not vary much for most of the single bug programs.

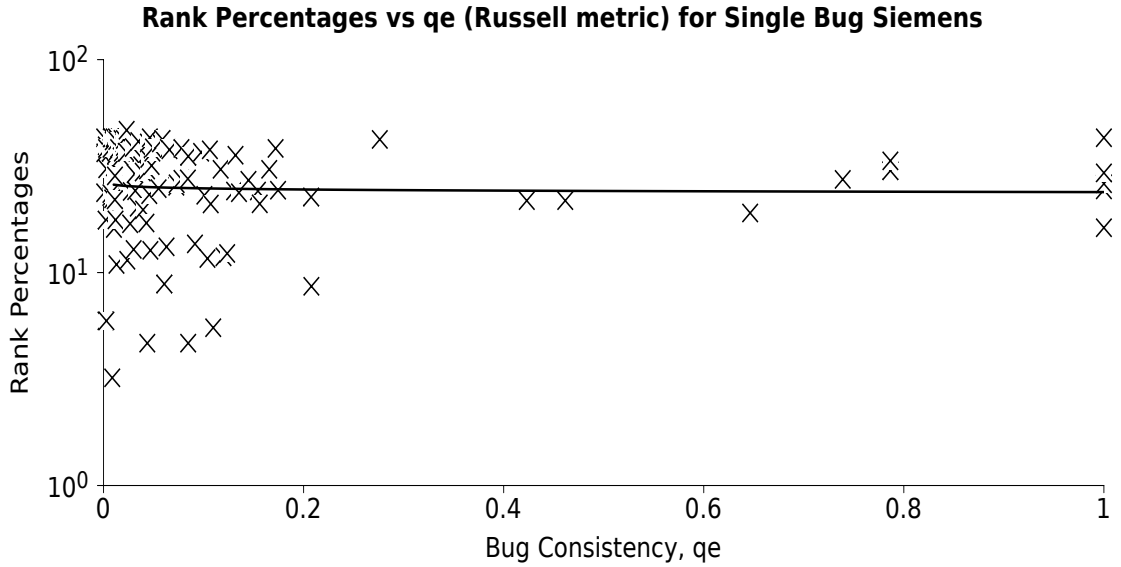

Figure 6.5: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the Russell metric

6.3.2 Subset of the Unix Test Suite

In this subsection, we plot the relationship of rank percentages using several spectra metrics evaluated on the subset of the Unix Test Suite programs. We evaluate a similar set of spectra metrics as in Subsection 6.3.1 to perform fair comparison on the metrics across different datasets. In the following figures, each point relates to the buggy statement of a typical program of 102 single bug of the subset of the Unix Test Suite.

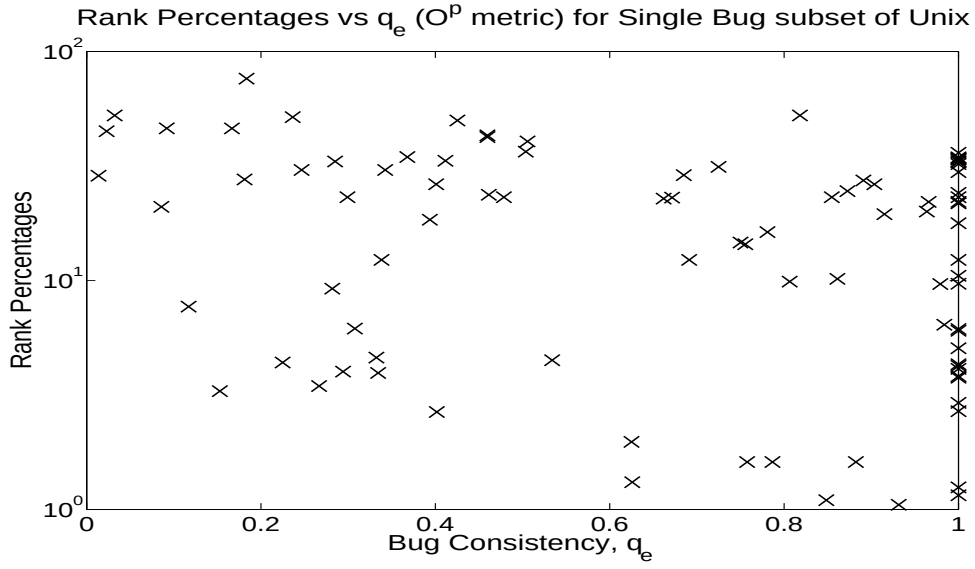

Figure 6.6: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the O^p metric

Figure 6.6 and Figure 6.7 show the relationship of rank percentages with respect to the bug consistency, q_e, for the O^p and Rogers metrics on the subset of the Unix Test Suite. These figures show different patterns of the points of rank percentages from what we observed earlier for the Siemens Test Suite in Subsection 6.3.1. Therefore, we do not plot any trend lines for the subset of the Unix Test Suite, and defer discussion of this behaviour to Subsection 6.3.3.

Figure 6.8 for the Russell metric shows a similar pattern of the points of rank percentages as Figure 6.5 for the Siemens Test Suite. Most of the points are in the rank percentages of 30%-60% for all the q_e values.

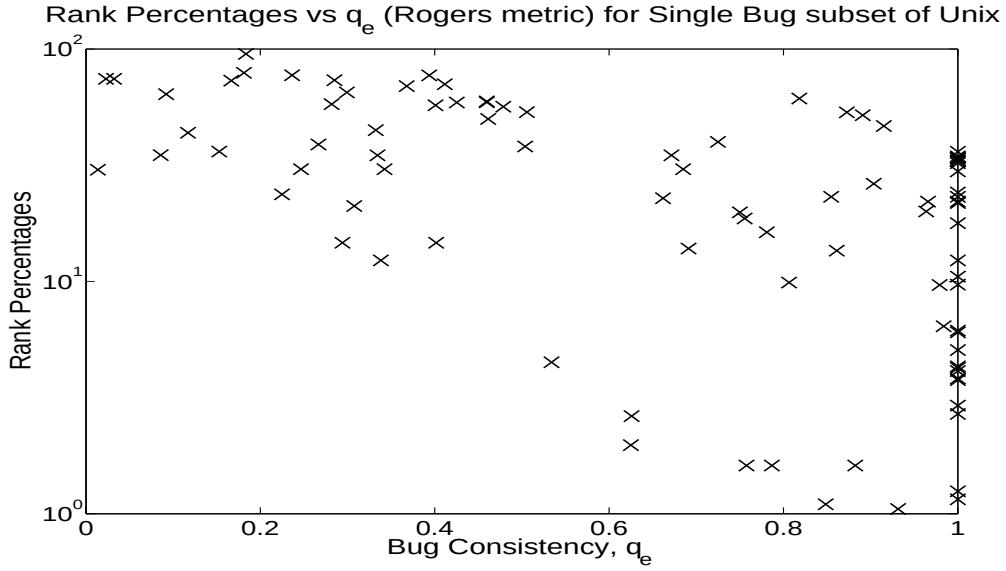

Figure 6.7: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the Rogers metric

We also plot similar relationships of rank percentages with respect to bug consistency, q_e, using a similar set of metrics, for the single bug programs of the Concordance and Space programs. These figures can be found in Appendix C. The appendix also shows the plots for other metrics on the Siemens Test Suite and the subset of the Unix Test Suite using spectra metrics, namely, Tarantula, Wong3, and Wong4.

In this study, we have shown that the relationship of rank percentages and bug consistency, q_e, in the Siemens Test Suite fits well with our hypothesis and the *Ideal Case* of Figure 6.1. However, we do not observe a similar trend for other benchmarks, such as the subset of the Unix Test Suite, Concordance, and Space programs. The points of the rank percentages on the subset of the Unix Test Suite and Space programs are spread out as the bug consistency, q_e, increases. One possible explanation for the different observation on the Siemens Test Suite from the other benchmarks is the test case design. The test cases in the Siemens Test Suite are generated by the researchers at Siemens [SIR, 2010] using

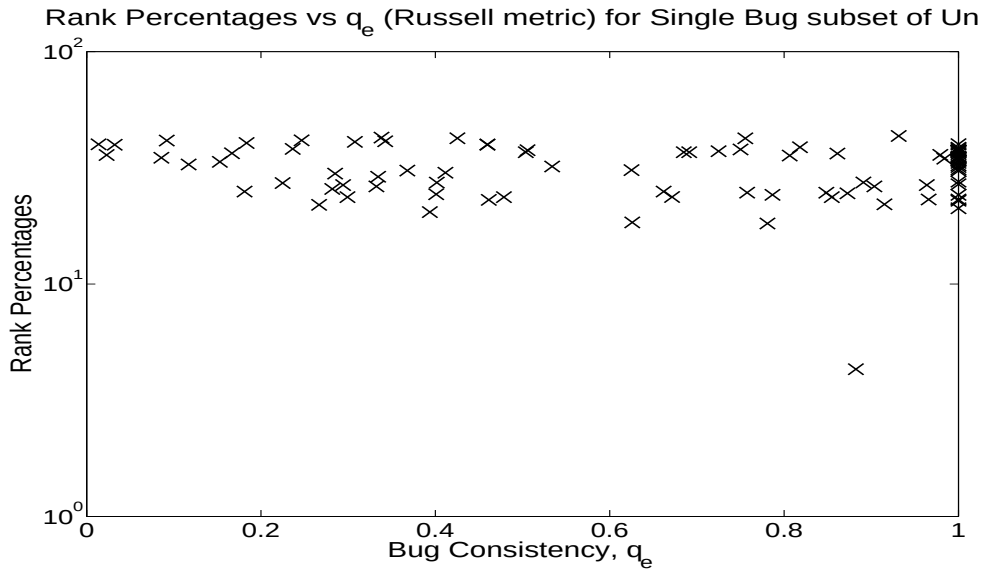

Figure 6.8: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the Russell metric

the category partition approach [Ostrand and Balcer, 1988] (detailed in Subsection 7.2.1). The researchers at Siemens also ensure that each executable statement of the program is executed by at least 30 test cases. For the subset of the Unix Test Suite, we observe a huge number of programs with ties (the buggy statement shares the same a_{ij} and metric value as non-buggy statements). This could affect bug localization performance, and is further explained in Subsection 6.3.3. For Concordance and Space, we cannot draw any strong conclusion from the rank percentages points, as we only have a small number of programs for these benchmarks.

6.3.3 Insights on the Bug Consistency, q_e with respect to the Rank Percentages

In Subsection 6.3.2, for the subset of the Unix Test Suite, we observe that some points with respect to the rank percentages (Figure 6.6 and Figure 6.7) do not agree with the hypothesis we have established in Section 6.2. A similar observation also applies to the Space programs in Figure C.13 and Figure C.14 of Subsection C.1.4.

Initially, we investigate the behaviour observed for the subset of the Unix Test Suite in Figure 6.6 and Figure 6.7. In these figures, when the q_e value is 1, we still have a huge number of rank percentages points especially between 10% and 50%. This is in contrast with our hypothesis in Figure 6.1. In our hypothesis, as the bug becomes more consistent (q_e value of 1), the buggy statement would have achieved the best bug localization performance, and be ranked top.

We investigate the programs in the subset of the Unix Test Suite dataset where the rank percentages are between 10% and 50% and the bug is consistent (q_e value is 1). We also investigate some of the programs that have lower rank percentages and the bug is consistent (q_e value is 1). This enables us to gain some understanding on the patterns of the programs observed in Figure 6.6 and Figure 6.7. We choose one of the metrics, O^p, to investigate the behaviour observed for the rank percentages on the subset of the Unix Test Suite. We propose to investigate the distribution of q_e values of some of the programs in the dataset, namely, Colv29, Checkeqv12, and Splinev14. The buggy statement of these programs has a q_e value of 1, but the bug localization performance evaluated with the O^p metric varies.

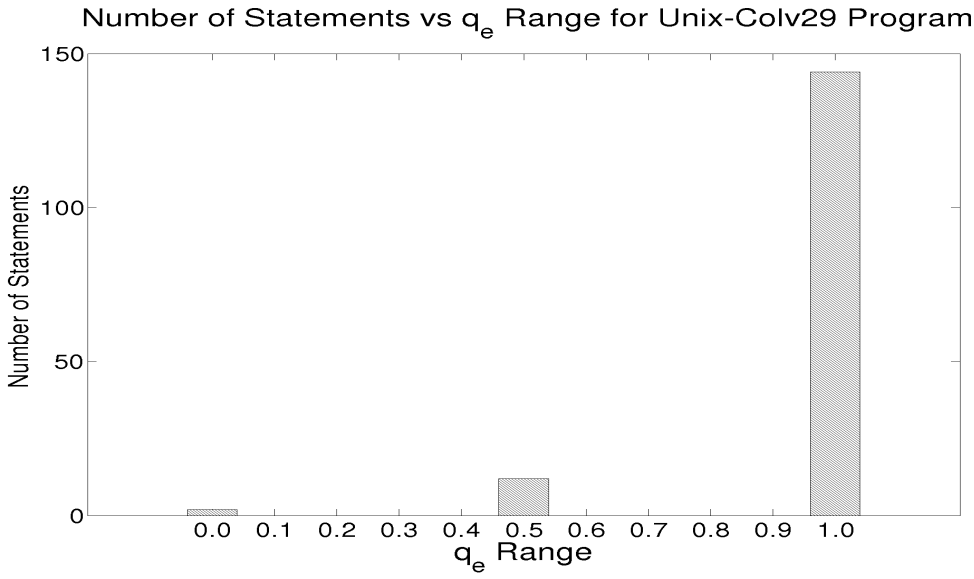

Figure 6.9: Number of Statements vs q_e Range for the subset of the Unix Test Suite-Colv29 using O^p metric

Initially, we choose to investigate Colv29 program. Figure 6.9 shows the histogram of the distribution of the q_e values (in ranges) for all the program statements of Colv29 of the subset of the Unix dataset. The q_e range of 0 in the x-axis refers to the number of program statements that have a q_e value greater than 0 and strictly lesser than 0.1. The q_e range of 0.1 refers to the number of program statements that have a q_e value of greater than 0.1 and strictly lesser than 0.2. The same equality condition for the q_e range of 0.1 also holds for the q_e range of 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9 respectively. The q_e range of 1.0 indicates that the number of program statements that have the q_e value equal to 1. By using the O^p metric, the rank percentages for this program is 33.22%.

There are 144 statements in the program that share the similar q_e value of 1 (q_e range of 1.0 in x-axis). We observe 104 of the 144 statements are ranked together with the buggy statement. These statements have identical a_{ij} values to the buggy statement. Some of

these statements are in the same block as the buggy statement, and so share the same metric value as the buggy statement. Since we use the *Mid* measure in the rank percentages (see Section 4.2), the average of the tied position range of the buggy statement is used. Therefore, we can observe poor performance of bug localization (rank percentages) for this particular program, even though the bug is consistent (q_e value is 1).

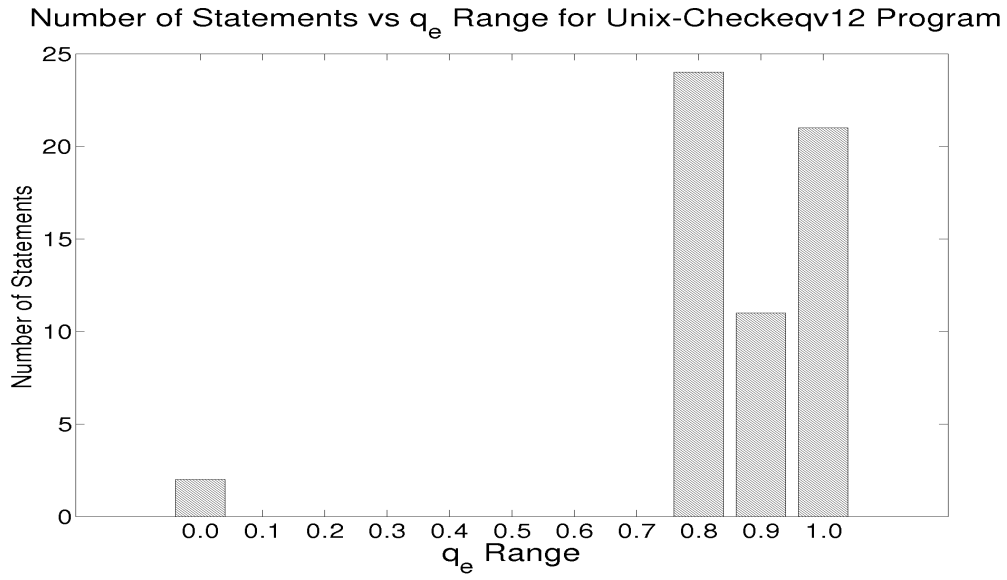

Figure 6.10: Number of Statements vs q_e Range for the subset of the Unix Test Suite-Checkeqv12 using O^p metric

We investigate the distribution of q_e values for all the statements of Checkeqv12 of the subset of the Unix Test Suite (Figure 6.10). This program performs slightly better than Colv29 on the O^p metric. The programmer needs to examine 4.31% of the program code in order to locate the bug. We observe 21 statements in the program that share the same q_e value of 1. These statements have 4 of the 21 statements ranked together with the buggy statement, and these four statements are in the same block as the buggy statement.

We also study another example of the distribution of q_e values for all the statements of Splinev14 from the subset of the Unix Test Suite in Figure 6.11. In this program, using the O^p metric, the programmer needs to examine 1.15% of the program code in order to locate the bug. We observe that there is only one statement having a q_e of 1 and ranked together with the buggy statement. This statement is in the same block of the buggy statement. As the number of statements in the program having a q_e value of 1 is smaller, the performance of bug localization for this program is better than the previous two programs (Colv29 and Checkeqv12) as expected. This result also holds for several other sensible metrics, including Zoltar and Wong3.

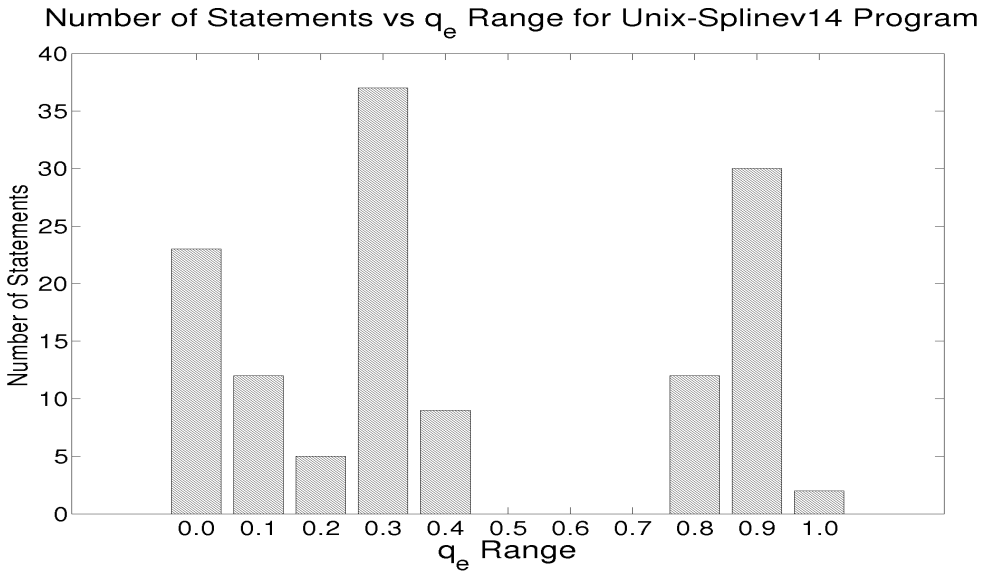

Figure 6.11: Number of Statements vs q_e Range for the subset of the Unix Test Suite-Splinev14 using O^p metric

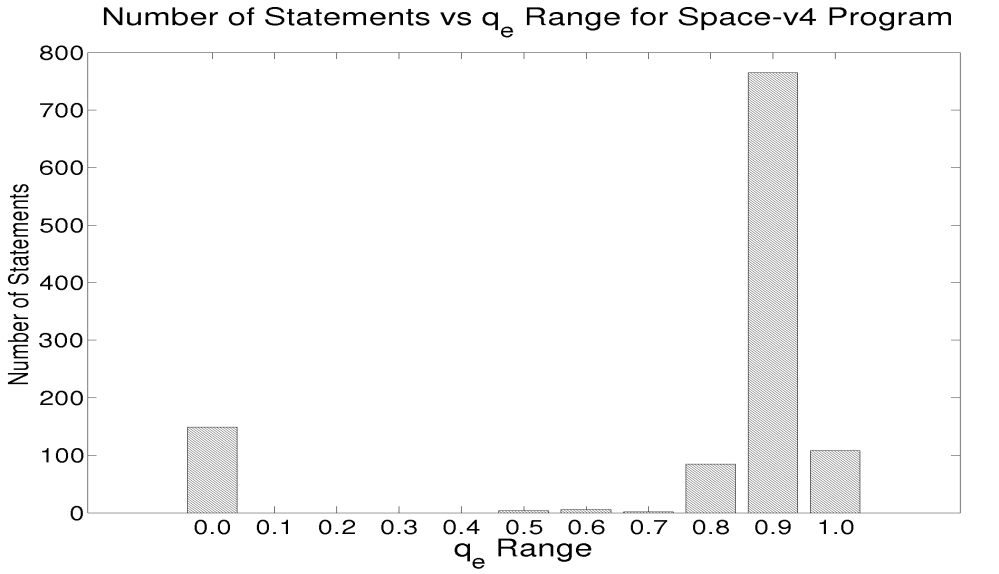

Figure 6.12: Number of Statements vs q_e Range for Spacev4 using O^p metric

We also investigate and gain understanding of the relationship of rank percentages and q_e for the Space program, Spacev4, with a q_e value of 1. By using the O^p metric, this program shows poor bug localization performance, with the rank percentages of 0.49%. The best rank percentages in the Space program is 0.06%. This program has a huge number of non buggy statements sharing the q_e value of 1 with the buggy statement. We plot the distribution of q_e values for all the statements of the Spacev4 in Figure 6.12. There are 108 statements that have a q_e value of 1. Out of the 108 statements, there are 69 that ranked together with the buggy statement. Therefore, we observe poor performance

of bug localization (rank percentages) for this particular program even though the bug is consistent (q_e of 1).

We have gained some insights from our investigation of the three examples of the subset of the Unix Test Suite programs and one example of the Space program. Even though q_e gives us information on how consistent the bug is, it does not necessarily guarantee that the bug will be ranked top in the ranking. Our hypothesis, established earlier in Figure 6.1, suits programs in the Siemens Test Suite (Subsection 6.3.1). We do not observe a huge number of non-buggy statements in the same block with the buggy statement, especially when the bug is consistent (q_e of 1).

We are also interested to investigate the types of bugs for the points observed in Figure 6.6 in the subset of the Unix Test Suite. We want to study the type of bugs where the bug consistency is 1 but the bug localization performance (rank percentages) is poor. There are 17 programs having such a condition and the bug localization performance for these programs is within the rank percentages range of 20% to 30%. 10 out of 17 programs have the bugs related either to constant or data type changes. A typical example for changes of constant is version 4 of the print_tokens2 program with the bug $id = 0$. This type of bug is harder to diagnose as it is executed by all the test cases. Bugs that occur in conditional statements are easier to diagnose; for example, certain test cases only execute the *then* or *else* condition. The other 7 programs in the figure that perform poor bug localization performance when the bug is consistent (q_e is 1) are related to the conditional statements. We expect these programs would give better bug localization performance. However, we investigate these 7 programs and there are a huge number of non-buggy statements having ties with the buggy statement. Some of these non-buggy statements share the same metric value, $MetValue$, as the buggy statement within the same block of programs. Therefore, poor bug localization performance (higher rank percentages) are observed for these programs.

The details of the type of bugs for all the programs in our benchmarks can be found in Appendix H.

6.3.4 Evaluating The Relationship of Rank Percentages vs Bug Consistency, q_e on the Multiple-bug Programs

We observe that the Kulczynski2 and Zoltar metrics perform the best in bug localization performance for the two-bug programs and three-bug programs of the Siemens Test Suite (Table 5.15 and Table 5.16 respectively). We plot the relationship between the rank percentages on these better performing metrics and the bug consistency, q_e, on the multiple-bug programs. We do not consider the subset of the Unix Test Suite in our plot, as we

have observed a huge number of ties that can affect the bug localization performance in Subsection 6.3.2.

Initially, we plot the relationship of the rank percentages on the multiple-bug programs of the Siemens Test Suite for the Kulczynski2 and Zoltar metrics (Figure 6.13 and Figure 6.14). Each point of X in these figures relates to one of the multiple-bug programs of the Siemens Test Suite. We consider the bug localization performance of multiple-bug programs, based on the buggy statement that has the highest rank in the ranking. Each point in these figures for the x-axis refers to the bug consistency, q_e, of the buggy statement of a particular multiple-bug program that is first found in the ranking by the programmer. In Figure 6.14, we observe most of the bugs of the three-bug programs of the Siemens Test Suite can be located within 10% of the program code using the Zoltar metric. The relationship observed from the points in both of the figures are closely similar to our observation for most metrics in the subset of the Unix Test Suite (Subsection 6.3.2). The points in these figures are spread out as the bug consistency, q_e, increases. In some of these points, the bug localization performance is affected due to the ties of non-buggy statements with the buggy statement. These observations are possibly due to the test cases design. For the multiple-bug programs, we picked any two or more single bug programs with different bugs and treated them as a multiple-bug program (detailed in Section 4.5). We do not redesign test cases for the multiple-bug programs, but use the same test suite from the Siemens Test Suite [SIR, 2010], originally designed for single bug programs.

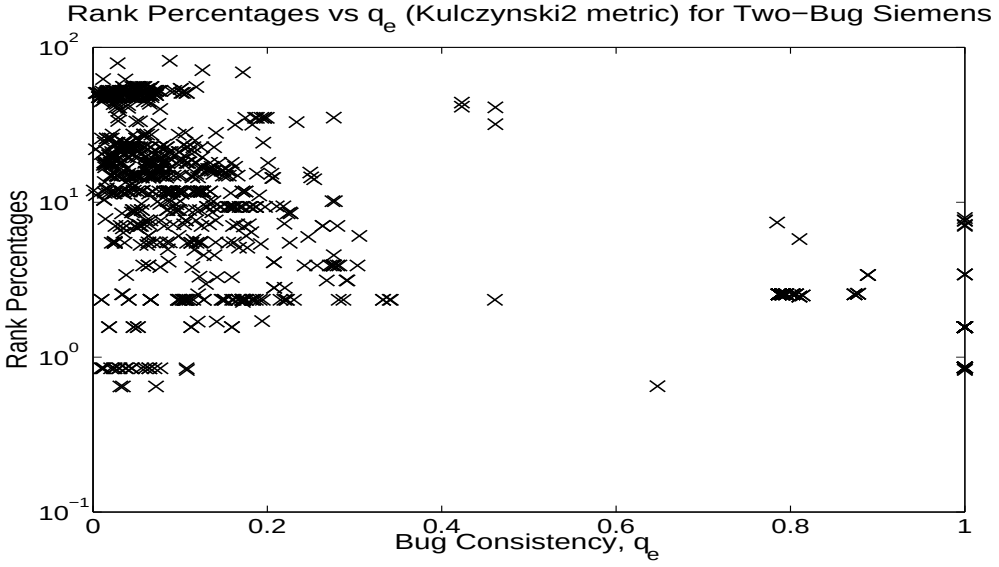

Figure 6.13: Rank Percentages vs q_e for the Two-bug Siemens Test Suite with respect to the Kulczynski2 metric

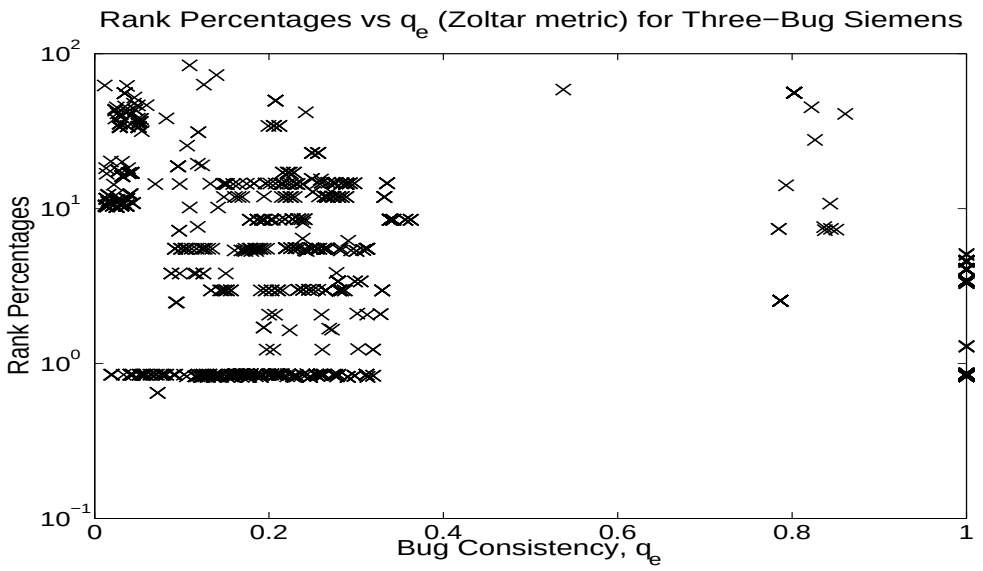

Figure 6.14: Rank Percentages vs q_e for the Three-bug Siemens Test Suite with respect to the Zoltar metric

We also plot the multiple-bug Space programs in Appendix C. As we have only 13 multiple-bug Space programs, we cannot make any conclusion from the observations on this benchmark.

6.4 Summary

In this chapter, we studied the relationship between bug localization performance and bug consistency, q_e. When the bug consistency, q_e, is approaching 1, we observe that the bug localization performance improves for better performing spectra metrics. However, not all test suites show this relationship. Bug localization performance is affected even though the bug was consistent (q_e is 1), as there are several statements in the same block as the buggy statement.

We also studied the relationship of rank percentages and bug consistency, q_e, on the multiple-bug programs of the Siemens Test Suite and Space. The points of rank percentages of the multiple-bug programs are spread out as the bug consistency, q_e, increases (similar to the single bug programs in the subset of the Unix Test Suite).

7

Bug Localization using Unique (Non-redundant) Test Cases

7.1 Introduction

A test suite often contains a large number of test cases. As a typical software project evolves, more test cases are added by programmers, and regression testing is always performed before a new software release, to ensure the software passes all the test cases. We wish to investigate any effect on the bug localization performance when we remove any redundant test cases having identical coverage patterns.

In the area of debugging, several studies have also looked into the idea of removing redundant test cases in the test suites [Harrold et al., 1993, Rothermel et al., 1998, Wong et al., 1998, Jones and Harrold, 2003, Heimdahl and George, 2004, Hao et al., 2005, Yu et al., 2008]. In these studies, the entire test suite containing all the test cases is referred to as the unreduced test suite. The test suite where the redundant test cases have been removed is known as a minimised test set or reduced test suite. In the previous chapters, we used the entire test suites of our benchmarks to evaluate bug localization performance. In this chapter, we perform thorough evaluation using non-redundant test cases to observe the effectiveness of bug localization performance. We use the term redundant test cases for the method that includes all the test cases, and the term unique test cases for the reduced test suite. We introduce the concept of using unique test cases to locate bugs with program spectra. We investigate the amount of redundant test cases in our benchmarks: the Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space programs. We then evaluate several spectra metrics using the unique test cases on our benchmarks. We also evaluate bug localization performance for unique test cases on multiple-bug programs of our benchmarks. We show the importance of using more unique test cases with respect to the bug localization performance for several spectra metrics.

7.2 Concept of Unique (Non-redundant) Test Cases

There are mainly two approaches to remove redundant test cases. Let t_A and t_B as the program spectra of test case A and test case B respectively. Each test t_A and t_B consists of a set of statements executed by the test case. In the first approach, t_A is considered redundant if $t_A \subseteq t_B$. This approach is referred to as statement-based reduction [Yu et al., 2008]. In the second approach, t_A is considered redundant if and only if t_A=t_B. This approach is referred to as vector-based reduction [Yu et al., 2008]. The first approach has been used by Harrold et al. [1993]. The second approach is used in our study.

Table 7.1: Code Fragment of Figure 1 from Yu et al. [2008]

		Test Cases							
		t1 3,3,5	t2 1,2,3	t3 3,2,1	t4 5,5,5	t5 5,3,4	t6 7,5,4	t7 2,1,3	t8 4,3,5
	`mid() { int x,y,z,m;`								
1:	`read("Enter 3 numbers:",x,y,z);`	•	•	•	•	•	•	•	•
2:	`m = z;`	•	•	•	•	•	•	•	•
3:	`if (y < z)`	•	•	•	•	•	•	•	•
4:	`if (x < y)`	•	•			•		•	•
5:	`m=y;`		•						
6:	`else if (x < z)`	•				•		•	•
7:	`m=y; // *** bug ***`	•						•	•
8:	`else`			•	•		•		
9:	`if (x > y)`			•	•		•		
10:	`m=y;`			•			•		
11:	`else if (x > z)`				•				
12:	`m=x;`								
13:	`print("Middle number is:",m);`	•	•	•	•	•	•	•	•
	`}` Pass/Fail Status	P	P	P	P	P	P	F	F

We illustrate these two approaches using a fragment of code given in Yu et al. [2008]. Table 7.1 shows a typical test coverage of a program with 8 tests and 13 statements. In this table, there are six pass test cases (t1–t6) and two fail test cases (t7–t8). The Pass and Fail status of the test cases are represented with P and F respectively. For the pass test cases, t1 $\neq$ t2 $\neq$ t3 $\neq$ t4. t5 $\subseteq$ t1 and t3 $=$ t6. For the fail test cases, t7 $=$ t8. In the case of vector-based reduction (used in our study), the unique test cases would be t1, t2, t3, t4, t5, t7. In the case of statement-based reduction, the unique test cases would be t1, t2, t3, t4, t7. In this study, we retain unique identical pass and fail test cases; that is, we treat pass and fail test cases independently.

Table 7.2: Breakdown of Unique Test Cases (on average) for the Subset of the Single Bug Programs

Program	Test Cases	Unique Test Cases	Unique Test Cases (%)
Concordance	372	132	35.48
replace	5542	2025	36.54
schedule	2650	469	17.69
schedule2	2710	664	24.50
tcas	1608	10	0.62
Space *Subset*	1103	895	81.14

7.2.1 Insights of Redundant Test Cases

We remove the redundant test cases from our benchmarks; Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space programs. In order to remove the redundant test cases and determine the unique test cases, all the test cases in the test suites have to be executed.

Due to the space constraint, we only show the information of the number of all test cases (redundant), number of unique test cases, and the percentages of the unique test cases (Unique Test Cases (%)) for several single bug programs of our benchmarks in Table 7.2. The figures for *Unique Test Cases* and *Unique Test Cases %* in this table are averaged across different program versions for each program. The proportion of unique test cases in *tcas* is 0.62%. This means that approximately 99.38% of test cases are redundant. There are huge number of redundant pass and fail test cases in the *tcas* program.

The Siemens Test Suite has a huge number of redundant test cases due to the test cases generated by the researchers at Siemens [SIR, 2010] using the category partition approach [Ostrand and Balcer, 1988]. This approach partitions and decomposes test specifications into several test cases. For example, a test specification of an array variable can be decomposed to test cases checking the *size of the array* and checking whether the *array is empty or full*. The researchers at Siemens also manually added more test cases in the test suite to ensure that each executable statement of the program is executed by at least 30 test cases.

For Concordance, the proportion of unique test cases per program (on average) is 35.48%. The redundant test cases are also observed for Space program. In this study, we use the *Subset* test cases of Space (consisting of 10 bins), which are randomly selected from the existing test suite of Space programs. The details of the *Subset* of Space can be found in Section 4.5. On average for a Space program in one bin, the proportion of unique test cases is 81.14%.

We also show the breakdown of unique test cases of other single bug and multiple-bug programs in Appendix D. The breakdown of the unique pass and fail test cases for each program version in the benchmarks can be found at the webpage (`http://www.cs.mu.oz.au/~leehj/unique/appendix.htm`).

7.3 Bug Localization Performance using Unique Test Cases

We evaluate the performance of bug localization based on the unique test cases (vector-based reduction approach) for the Siemens Test Suite and the subset of the Unix Test Suite, Concordance, and Space programs.

Yu et al. evaluate both the statement-based and vector-based reduction approaches to study the effect of bug localization performance [Yu et al., 2008]. In their study, Yu et al. use the subsets of test cases before removing redundant test cases. They use the subsets of test cases in the Siemens Test Suite and Space to study the effect of bug localization performance with respect to the different test suite sizes. In our study, we use the entire test cases of the test suites before removing any identical test cases for pass and fail test cases. We evaluate the unique test cases using several better performing spectra metrics (which have been found in Chapter 5), such as O^p, Zoltar, and Wong3. Yu et al. found that the vector-based reduction approach gives an improvement in bug localization performance as compared to the statement-based reduction approach for the Siemens Test Suite and Space. In their evaluation, they use Tarantula, Jaccard, and Ochiai metrics. They report their evaluation based on all the Siemens Test Suite and Space programs regardless of the number of bugs in these programs. We report the figures according to the breakdown of the number of bugs in the programs: single bug, two-bug, and three-bug programs.

7.3.1 Single Bug Programs

Table 7.3: Average Rank Percentages for Redundant and Unique Test Cases of the Single Bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	Redundant	Unique	p-value
O, O^p	17.86	17.79	0.8117
Zoltar	18.23	18.00	0.6501
Kulczynski2	19.06	18.56	0.2053
McCon	19.06	18.56	0.2053
JacCube	20.06	18.67	0.0032
M2	20.12	18.88	0.0005
Continued on next page			

Table 7.3 – continued from previous page

Metric	Redundant	Unique	p-value
Wong3	18.19	18.90	1
Ochiai	21.63	19.65	0.0082
Wong4	18.93	20.78	1
Jaccard	23.64	21.58	0.0245
Pearson	23.56	22.19	0.0013
AMean	23.66	22.38	0.0010
Ample2	24.08	23.33	0.0874
Rogot2	24.88	23.57	0.0047
Tarantula	27.09	25.91	0.0158
CBI Log	26.80	28.11	0.6631
Russell	30.02	30.02	1
Binary	30.02	30.02	1
Ample	30.16	30.04	0.5625
Overlap	32.23	32.23	1

Table 7.4: Average Rank Percentages for Redundant and Unique Test Cases of the Single Bug Concordance Programs

Metric	Redundant	Unique	p-value
O,O^p	10.11	10.14	0.9632
Zoltar	10.11	10.14	0.9632
Kulczynski2	10.21	10.20	0.6054
McCon	10.21	10.20	0.6054
JacCube	10.43	10.22	0.0987
M2	10.97	10.40	0.2113
Ochiai	11.19	10.51	0.1006
Wong3	10.15	11.19	0.9849
Jaccard	17.68	12.35	0.2771
Wong4	11.35	12.75	0.3937
AMean	18.90	13.76	0.0173
Pearson	18.42	13.91	0.5
Ample2	18.05	14.68	0.7908
Tarantula	20.03	15.81	0.5

Continued on next page

Table 7.4 – continued from previous page

Metric	Redundant	Unique	p-value
Rogot2	19.24	19.73	0.8886
Overlap	21.03	21.03	1
Russell	21.03	21.03	1
Binary	21.03	21.03	1
CBI Log	22.63	25.71	0.8532
Ample	27.53	40.06	0.9929

Table 7.5: Average Rank Percentages for Redundant and Unique Test Cases of the Single Bug Space Programs (on average of 10 bins)

Metric	Redundant	Unique	p-value
O,O^p	1.64	1.63	0.5
Wong3	1.65	1.74	0.9093
Zoltar	1.80	1.78	0.5
JacCube	1.90	1.88	0.5
M2	1.92	1.89	0.8145
Kulczynski2	2.07	2.01	0.2113
McCon	2.07	2.01	0.2113
Ochiai	2.26	2.23	0.8145
Wong4	1.64	2.54	0.8618
Rogot2	2.67	2.90	0.8193
Pearson	2.72	2.96	0.9093
Ample2	2.68	3.15	0.9704
AMean	2.93	3.15	0.7052
Jaccard	3.18	3.36	0.8193
Tarantula	6.31	6.42	0.5472
Ample	6.56	8.24	0.9704
CBI Log	6.65	8.46	0.7219
Russell	17.59	17.59	1
Binary	17.59	17.59	1
Overlap	18.31	18.31	1

Table 7.3, Table 7.4, and Table 7.5 show the bug localization performance (average rank percentages) using unique test cases on all the single bug programs from the Siemens Test Suite and the subset of the Unix Test Suite, Concordance, and Space. We also perform a one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] on the benchmarks to check the statistical significance of our hypothesis - *the bug localization performance using the unique test cases improves as compared to the bug localization performance using the redundant test cases*. We report the p-value [Rice, 1989] for our hypothesis in these tables. We choose p-value of 0.05 for our hypothesis. For the Space programs, we only perform the statistical test on one of the 10 bins in the *Subset* of Space programs, as the bug localization performance across these 10 bins is very similar (shown in Figure 5.13 of Subsection 5.8.1).

When using the better performing metrics such as O and O^p, there is a slight improvement in bug localization performance using unique test cases as compared to using redundant test cases on the Siemens Test Suite and the subset of the Unix Test Suite, and Space programs. Even though these metrics show a slight improvement in bug localization performance using unique test cases, we observe larger p-values for these metrics. p-values greater than 0.05 indicate that the improvement in bug localization performance using unique test cases as compared to using redundant test cases for these metrics is not statistically significant. We observe a slight drop in the effectiveness of the bug localization performance for O and O^p metrics on Concordance by using unique test cases as compared to using redundant test cases.

Metrics shown at the bottom of the Table 7.3, such as Rogot2 and Tarantula show statistically significant improvement of bug localization performance using unique test cases as compared to using redundant test cases. These metrics have lower p-values as compared to the better performing metrics such as O and O^p. For example, the Tarantula metric in Table 7.3 shows statistically significant improvement of bug localization performance (p-value less than 0.05) using unique test cases as compared to using redundant test cases. Yu et al. found that bug localization performance using unique test cases on the Tarantula metric improves as compared to using redundant test cases of the Siemens Test Suite and Space programs [Yu et al., 2008]. However, metric such as Tarantula is not very useful for bug localization. We observe the programmer needs to examine more program code (higher average rank percentages in Table 7.3, Table 7.4, and Table 7.5) to locate bugs using the Tarantula metric as compared to using the better performing metrics such as O and O^p. Generally for most of the single bug benchmarks, we do not observe any degradation of bug localization performance for better performing metrics by using unique test cases.

7.3.2 Multiple-bug Programs

We evaluate our proposed unique test cases approach on multiple-bug programs. We evaluate on the two-bug and three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite in Table 7.6 and Table 7.7 respectively. We also evaluate the unique test cases approach on multiple-bug Space programs in Table 7.8. In these tables, we report the p-value of our hypothesis, which is defined in Subsection 7.3.1.

Table 7.6: Average Rank Percentages for Redundant and Unique Test Cases of the Two-bug Programs of the Siemens Test Suite and the subset of the Unix Test Suite

Metric	Redundant	Unique	p-value
Pearson	21.09	20.04	<0.05
Ample2	21.37	20.11	<0.05
AMean	20.97	20.13	<0.05
Jaccard	21.10	20.29	<0.05
Rogot2	21.78	20.40	<0.05
Wong4	20.78	20.58	0.9998
Kulczynski2	19.53	20.89	0.2171
McCon	19.53	20.89	0.2171
Ochiai	20.18	20.93	0.8003
CBI Log	22.29	21.13	<0.05
Zoltar	20.52	21.16	0.9848
Tarantula	23.13	21.61	<0.05
Wong3	22.54	22.33	1
M2	20.80	22.35	0.7035
JacCube	20.51	22.37	0.9080
O^p	22.84	23.19	0.9443
Ample	24.49	23.67	<0.05
O	24.95	25.05	0.9240
Russell	32.10	32.44	1
Binary	34.02	34.02	1
Overlap	34.02	34.08	0.7305

Table 7.7: Average Rank Percentages for Redundant and Unique Test Cases of the Three-bug Programs of the Siemens Test Suite and the subset of the Unix Test Suite

Metric	Redundant	Unique	p-value
Wong4	22.60	22.29	0.0583
Ample2	22.56	22.86	0.7748
CBI Log	23.37	22.87	0.0004
Pearson	22.31	23.19	1
Rogot2	22.53	23.21	0.9983
AMean	22.24	23.28	1
Ample	23.88	23.78	0.1698
Zoltar	23.49	25.04	1
Kulczynski2	21.94	25.16	1
McCon	21.94	25.16	1
Ochiai	22.60	25.52	1
Jaccard	23.12	25.73	1
Wong3	25.24	26.21	1
M2	24.97	26.79	1
O^p	25.33	26.90	1
Tarantula	27.23	26.94	0.2674
JacCube	24.86	27.03	1
O	29.29	29.48	1
Russell	28.67	30.05	1
Binary	32.43	32.43	1
Overlap	38.34	38.42	0.9207

Table 7.8: Average Rank Percentages for Redundant and Unique Test Cases of the Multiple-bug Space Programs (on average of 10 bins)

Metric	Redundant	Unique	p-value
Zoltar	2.67	2.42	0.4
JacCube	2.75	2.47	0.3946
O	2.80	2.55	0.5
O^p	2.80	2.55	0.5
Wong3	2.80	2.55	0.5
M2	2.83	2.58	0.2919

Continued on next page

Table 7.8 – continued from previous page

Metric	Redundant	Unique	p-value
Kulczynski2	2.91	2.63	0.5
McCon	2.91	2.63	0.5
Ochiai	3.13	2.77	0.3
Wong4	3.09	2.80	0.2092
Jaccard	3.70	3.25	0.0907
Rogot2	3.75	3.26	0.6054
Pearson	3.84	3.41	0.6054
Ample2	3.84	3.51	0.5
AMean	4.00	3.59	0.6054
CBI Log	4.21	3.80	0.6063
Tarantula	4.85	4.36	0.3051
Ample	8.82	8.39	0.2919
Binary	19.62	17.85	0.4392
Russell	19.62	17.85	0.4392
Overlap	19.87	18.07	0.4645

In Table 7.6 and Table 7.7, we observe a slight improvement in bug localization performance on unique test cases of the multiple-bug Siemens Test Suite and the subset of the Unix Test Suite programs, for better performing metrics such as Pearson and Wong4. In Table 7.8, we observe an improvement in bug localization performance for the better performing Zoltar metric for multiple-bug Space programs with the improvement of average rank percentages of 0.25%. However, the latter improvement observed in Table 7.8 is not statistically significant (p-value of 0.4). The Tarantula metric shows improvement in bug localization performance for multiple-bug programs using the unique test cases approach as compared to using the redundant test cases approach. The improved bug localization performances for the Tarantula metric in Table 7.6, Table 7.7, and Table 7.8 are 1.52%, 0.29%, and 0.49% respectively. However, the improved bug localization performance of using unique test cases with the Tarantula metric is only statistically significant (p-value less than 0.05) for the two-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite in Table 7.6. In our evaluation, using unique test cases on the multiple-bug programs also does not show any degradation of the bug localization performance for better performing metrics, namely Pearson in Table 7.6, Wong4 in Table 7.7, and Zoltar in Table 7.8.

7.4 Study of Varying the Number of Unique Test Cases with respect to Bug Localization Performance

In this section, we investigate bug localization performance by varying the number of unique test cases. We want to observe whether using more unique test cases can help to improve bug localization performance. We vary the percentage of unique test cases and observe the trend in bug localization performance.

Our study in this section is related to the previous finding in Subsection 5.7.3 where we observe the relationship of bug localization performance (using model program $ITE2_8$) by varying the number of fail test cases. The number of pass test cases also changes by varying the number of fail test cases. However, we observe no great improvement of bug localization performance using 4, 5, and 6 fail tests.

In this section, for each percentage of unique test cases, we randomly perform test selection from the pool of unique test cases. We perform test selection from the pool of unique test cases (for pass, fail, and both pass and fail test cases, respectively). To avoid any bias in test case selection, the random selection of unique test cases is performed ten times for different percentages of unique test cases. These random selections of respective percentages of unique test cases generate ten representative sets of randomly selected unique test cases. For each percentage of unique test cases, we report the bug localization performance by taking the average of the bug localization performance across the ten representative sets of randomly selected unique test cases. We report the results according to the breakdown of the benchmarks; Siemens Test Suite and the subset of the Unix Test Suite, Concordance, and Space. Evaluation is performed on single bug and multiple-bug programs of our benchmarks. We establish a hypothesis, that *the bug localization performance using all the unique test cases (100%) will improve as compared to using only 10% of the unique test cases*. We use one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] and report p-value to check the statistical significance of the hypothesis.

Initially, we evaluate the different percentages of unique pass test cases for the single bug programs of the Siemens Test Suite and the subset of the Unix Test Suite in Table 7.9. In each percentage of the unique pass test cases evaluated, the number of fail test cases remains the same. For example, if we have 100 unique pass and fail test cases, for 10%, 20%, 30%, 40%, 50%, and 80%, the effective unique pass test cases we evaluate are 10, 20, 30, 40, 50, and 80 unique pass test cases respectively. The fail test cases still contain the existing 100 unique fail test cases. The column 100% of the table refers to using all the unique pass and fail test cases. The bug localization performance for the Column Unique (100%) of Table 7.9 is equivalent to the average rank percentages figures we obtain in the Column Unique of Table 7.3. In this table, we report the p-value of our hypothesis. As

we only vary the percentages of the unique pass test cases, our hypothesis is that *the bug localization performance using all the unique test cases (100%) will improve as compared to using only 10% of the unique pass test cases.*

Table 7.9: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass Test Cases - Single Bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
O,O^p	20.37	19.60	18.83	18.45	18.29	18.00	17.79	<0.05
Zoltar	20.39	19.62	18.96	18.62	18.47	18.23	18.00	<0.05
Kulczynski2	20.88	20.12	19.47	19.13	18.99	18.76	18.56	<0.05
McCon	20.88	20.12	19.47	19.13	18.99	18.76	18.56	<0.05
JacCube	20.83	20.21	19.49	19.14	19.01	18.79	18.67	<0.05
M2	20.97	20.34	19.64	19.31	19.17	19.00	18.88	<0.05
Wong3	20.67	19.91	19.48	19.31	19.22	18.84	18.90	<0.05
Ochiai	21.28	20.70	20.13	19.87	19.81	19.72	19.65	<0.05
Wong4	24.30	22.74	21.82	21.27	21.09	20.92	20.78	<0.05
Jaccard	21.90	21.68	21.31	21.18	21.29	21.60	21.58	<0.05
Pearson	25.48	24.40	23.57	22.94	22.80	22.46	22.19	<0.05
AMean	25.75	24.71	23.93	23.22	23.03	22.65	22.38	<0.05
Ample2	26.36	25.40	24.63	23.96	23.81	23.42	23.33	<0.05
Rogot2	26.89	25.98	25.05	24.68	24.31	23.97	23.57	<0.05
Tarantula	28.73	27.93	27.19	26.64	26.42	26.11	25.91	<0.05
CBI Log	26.72	26.91	27.23	27.27	27.49	27.80	28.11	1
Russell	30.02	30.02	30.02	30.02	30.02	30.02	30.02	1
Binary	30.02	30.02	30.02	30.02	30.02	30.02	30.02	1
Ample	34.11	32.83	31.56	31.01	30.75	30.27	30.04	<0.05
Overlap	33.92	33.43	33.06	32.75	32.55	32.37	32.23	<0.05

For better performing metrics such as O^p, we observe that the bug localization performance gradually improves as we evaluate using larger number of unique pass test cases. We observe improved average rank percentages of 2.58% for this metric using all the unique pass test cases as compared to using 10% of the unique pass test cases. For most of the metrics, as the number of unique of pass test cases increases, the bug localization performance of the metric converges to the bug localization performance for Column

Unique (100%) of Table 7.9. We observe that the CBI Log metric shows the bug localization performance to worsen as we evaluate using a larger number of unique pass test cases. In this table, the Russell and Binary metrics do not show any improvement in bug localization performance for different percentages of unique pass test cases. Russell metric ranks statements based on a_{ef} (refer to Russell metric in Table 2.3). Varying the number of unique pass test cases for the Russell metric does not vary the ranking of statements, especially of buggy statements. The same goes for the Binary metric (refer to the metric in Table 2.3), where the metric relies on the fail test cases. Therefore, the bug localization performance of Russell and Binary metrics stay the same across the different percentages of unique pass test cases. Except for the CBI Log, Russell, and Binary metrics, the p-value of our hypothesis for all the metrics is less than 0.05. This indicates that for most of the metrics, the improvement in bug localization performance using all the unique test cases as compared to using only 10% of the unique pass test cases is statistically significant with confidence greater than 95%.

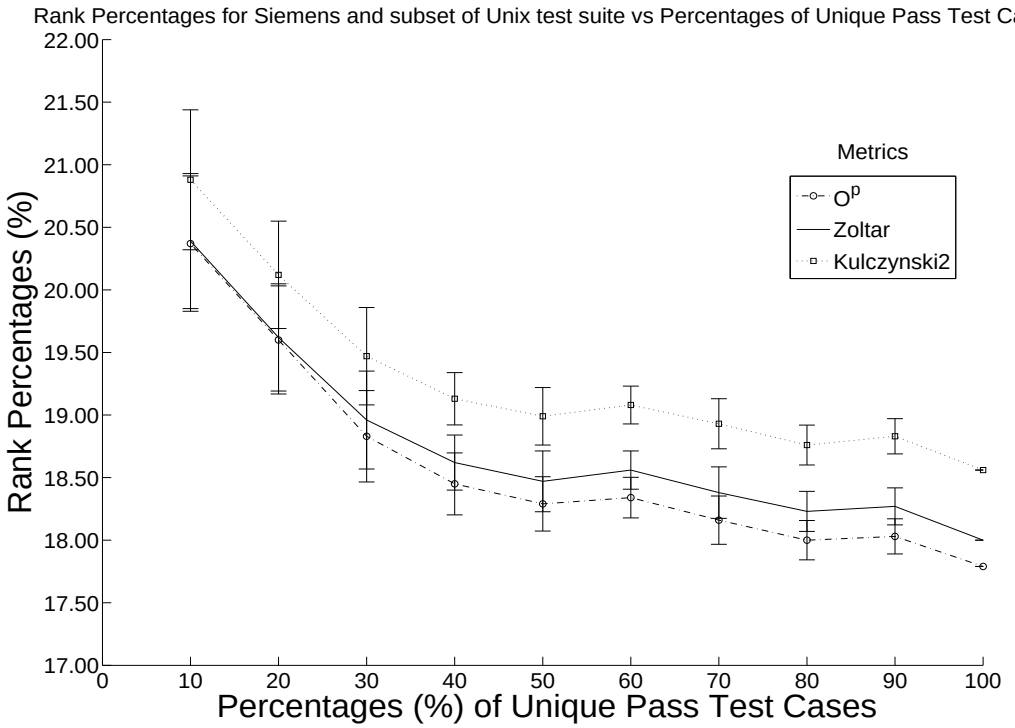

Figure 7.1: Average Rank Percentages (on average) for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite vs Percentages of the Unique Pass Test Cases

We plot the average of the bug localization performance (O^p, Zoltar, and Kulczynski2 metrics) of the ten representative sets of unique test cases (randomly selected) for the different percentages of the unique pass test cases (see Figure 7.1). We observe that bug localization performance for the different percentages of unique pass test cases converges

to the bug localization performance using all the unique test cases. However, we observe some fluctuation in the effectiveness of bug localization performance for these metrics. The bug localization performance for these metrics has slightly worsened, especially from the range of 50% to 60% of unique pass test cases. Using these metrics, we observe that the programmer needs to examine more program code in order to locate the bug using 60% of the unique pass test cases as compared to using 50% of the unique pass test cases. Similar worsened bug localization performance has also been observed from the range of 80% to 90% of the unique pass test cases.

In Figure 7.1, we plot the standard deviation (error bars) of the performance of bug localization for the ten representative sets of unique test cases (randomly selected) for the different percentages of the unique pass test cases. The width of the error bars in the figure indicates that the variation of bug localization performance of the ten representative sets of randomly selected percentages of the unique pass test cases. The error bar for Kulczynski2 overlaps with the O^p and Zoltar metrics in the range of 10%, 20%, and 30% of unique pass test cases. As the range of percentage of unique pass test cases goes beyond 30%, only the error bar of O^p overlaps with the error bar of the Zoltar metric.

Table 7.10: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Fail Test Cases - Single Bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
O,O^p	23.97	21.52	20.51	19.77	19.34	18.53	17.79	<0.05
Zoltar	24.13	21.79	20.88	20.13	19.67	18.75	18.00	<0.05
Kulczynski2	24.13	21.82	21.02	20.33	19.96	19.18	18.56	<0.05
McCon	24.13	21.82	21.02	20.33	19.96	19.18	18.56	<0.05
JacCube	24.28	21.98	21.12	20.52	20.13	19.29	18.67	<0.05
M2	24.28	22.01	21.29	20.70	20.32	19.57	18.88	<0.05
Wong3	27.16	24.26	23.15	22.24	21.51	20.28	18.90	<0.05
Ochiai	24.80	22.97	22.27	21.55	21.16	20.34	19.65	<0.05
Wong4	27.87	26.42	25.05	24.26	22.02	21.00	20.78	<0.05
Jaccard	26.18	25.01	24.35	23.75	23.22	22.39	21.58	<0.05
Pearson	25.74	24.60	24.02	23.55	23.29	22.74	22.19	<0.05
AMean	25.58	24.38	23.89	23.43	23.22	22.77	22.38	<0.05
Ample2	25.83	24.90	24.42	24.10	23.95	23.54	23.33	<0.05
Rogot2	27.12	26.03	25.46	25.07	24.79	24.14	23.57	<0.05
Tarantula	26.93	26.62	26.45	26.33	26.25	26.07	25.91	<0.05

Continued on next page

Table 7.10 – continued from previous page

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
CBI Log	37.19	32.17	31.15	30.09	29.34	28.71	28.11	<0.05
Russell	35.32	33.22	32.40	31.77	31.30	30.59	30.02	<0.05
Binary	35.32	33.22	32.40	31.77	31.30	30.59	30.02	<0.05
Ample	34.20	32.50	31.84	31.51	31.14	30.51	30.04	<0.05
Overlap	36.19	34.77	34.23	33.72	33.37	32.73	32.23	<0.05

In Table 7.10, we observe the bug localization performance for single bug programs from the Siemens Test Suite and the subset of the Unix Test Suite, with respect to different percentages of unique fail test cases. In each percentage of the unique fail test cases evaluated, the number of pass test cases remains the same. As we only vary the percentages of the unique fail test cases, our hypothesis is *the bug localization performance using all the unique test cases (100%) improves as compared to using only 10% of the unique fail test cases*.

For most of the metrics in this table, we observe that the bug localization performance gradually improves as we evaluate using larger number of unique fail test cases. As we evaluate using larger number of unique fail test cases, the bug localization performance for better performing metrics converges to the bug localization performance of using all the unique test cases (Column Unique (100%) of Table 7.10). We do not observe any fluctuation in the bug localization performance by varying only the unique fail test cases. As we evaluate using larger number of unique fail test cases, the buggy statement is executed more frequently. This helps to indicate that the buggy statement is likely to be buggy, allowing the programmer to locate the bug more effectively. Several studies [Wong et al., 2010, Debroy et al., 2010, Xie et al., 2010] discuss the importance of using fail test cases in helping to locate bugs; the details of these studies can be found in Chapter 2. In this table, we observe the Russell and Binary metrics also show improvement in bug localization performance when varying only the number of unique fail test cases. The p-value of our hypothesis for all the metrics is less than 0.05. This indicates that for all the metrics, the improvement of the bug localization performance using all the unique test cases as compared to using only the 10% of the unique fail test cases is statistically significant with confidence greater than 95%.

We plot the average of the bug localization performance (O^p, Zoltar, and Kulczynski2 metrics) of the ten representative sets of unique test cases (randomly selected) for different percentages of the unique fail test cases (see Figure 7.2). This figure shows improvement in bug localization performance for the metrics as we evaluate using larger number of

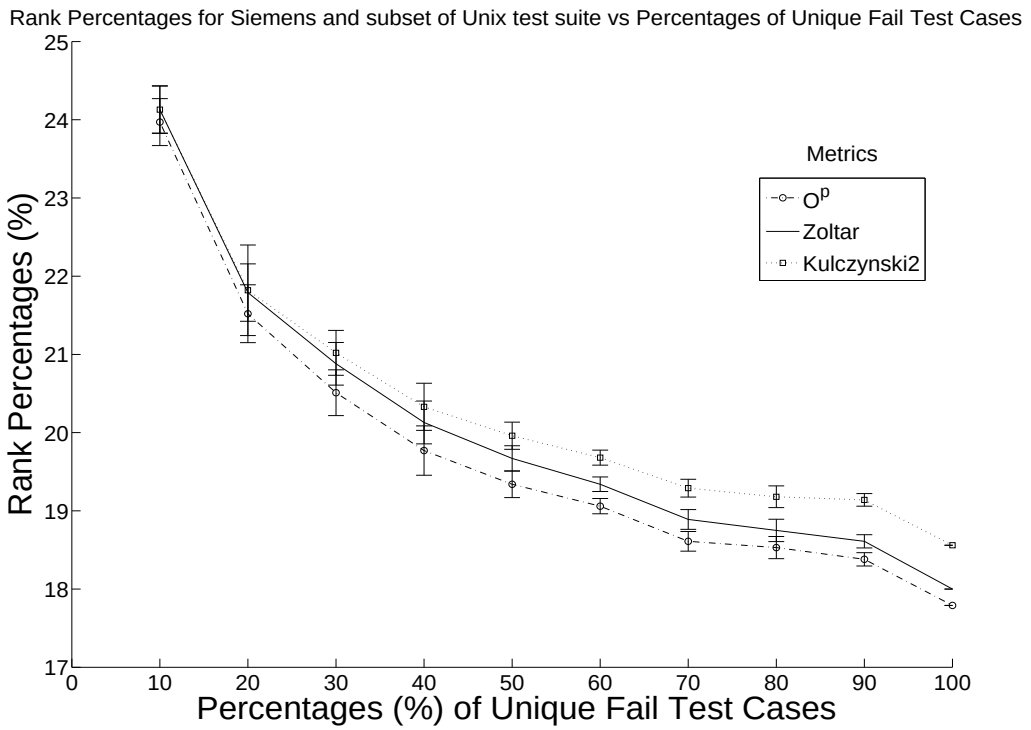

Figure 7.2: Average Rank Percentages (on average) for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite vs Percentages of the Unique Fail Test Cases

unique fail test cases. We observe that the width of error bars for these metrics are smaller than the error bars for the previous evaluation on unique pass test cases in Figure 7.1. This indicates less variation of bug localization performance in the ten representative sets of unique test cases for different percentages of unique fail test cases.

Table 7.11: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass and Fail Test Cases - Single Bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
O, O^p	26.02	23.20	21.47	20.18	19.61	18.58	17.79	<0.05
Zoltar	26.10	23.29	21.68	20.36	19.84	18.83	18.00	<0.05
Kulczynski2	26.17	23.40	21.89	20.58	20.12	19.28	18.56	<0.05
McCon	26.17	23.40	21.89	20.58	20.12	19.28	18.56	<0.05
JacCube	26.18	23.47	21.85	20.71	20.22	19.28	18.67	<0.05
M2	26.21	23.54	22.01	20.90	20.40	19.54	18.88	<0.05
Wong3	26.69	23.63	22.18	21.23	20.89	19.94	18.90	<0.05
Ochiai	26.64	24.18	22.82	21.55	21.11	20.30	19.65	<0.05

Continued on next page

Table 7.11 – continued from previous page

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
Wong4	29.76	27.89	25.70	24.60	22.14	21.09	20.78	<0.05
Jaccard	27.47	25.65	24.31	23.28	22.90	22.26	21.58	<0.05
Pearson	27.80	26.29	24.96	24.05	23.58	22.79	22.19	<0.05
AMean	27.87	26.39	25.06	24.14	23.67	22.89	22.38	<0.05
Ample2	27.96	26.65	25.44	24.62	24.23	23.58	23.33	<0.05
Rogot2	29.73	28.13	26.87	25.62	25.18	24.36	23.57	<0.05
Tarantula	29.01	28.41	27.44	26.88	26.54	26.09	25.91	<0.05
CBI Log	35.64	30.72	29.77	29.08	28.64	28.47	28.11	<0.05
Russell	35.32	33.24	32.25	31.60	31.25	30.54	30.02	<0.05
Binary	35.32	33.24	32.25	31.60	31.25	30.54	30.02	<0.05
Ample	35.84	34.15	32.99	32.05	31.54	30.69	30.04	<0.05
Overlap	36.50	35.18	34.54	33.92	33.55	32.82	32.23	<0.05

In Table 7.11, we observe the bug localization performance for single bug programs from the Siemens Test Suite and the subset of the Unix Test Suite with respect to the different number of unique pass and fail test cases for the Siemens Test Suite and the subset of the Unix Test Suite. In this table, our hypothesis is that *the bug localization performance using all the unique test cases (100%) improves as compared to using only 10% of the unique pass and fail test cases*.

We also observe a similar improvement of bug localization performance for all the metrics as we use more unique pass and fail test cases. As we use larger number of unique pass and fail test cases, we also observe similar convergence of bug localization performance to the bug localization performance of using all the unique test cases in Table 7.11. The p-value of our hypothesis for all the metrics is less than 0.05. Therefore, for all the metrics, the improvement of the bug localization performance using all the unique test cases as compared to using only the 10% of the unique pass and fail test cases is statistically significant with confidence greater than 95%.

In Figure 7.3, we plot the average of the bug localization performance (O^p, Zoltar, and Kulczynski2 metrics) of the ten representative sets of unique test cases (randomly selected) for different percentages of the unique pass and fail test cases. The trend of improvement on the bug localization performance observed in this figure is very similar to Figure 7.1 and Figure 7.2.

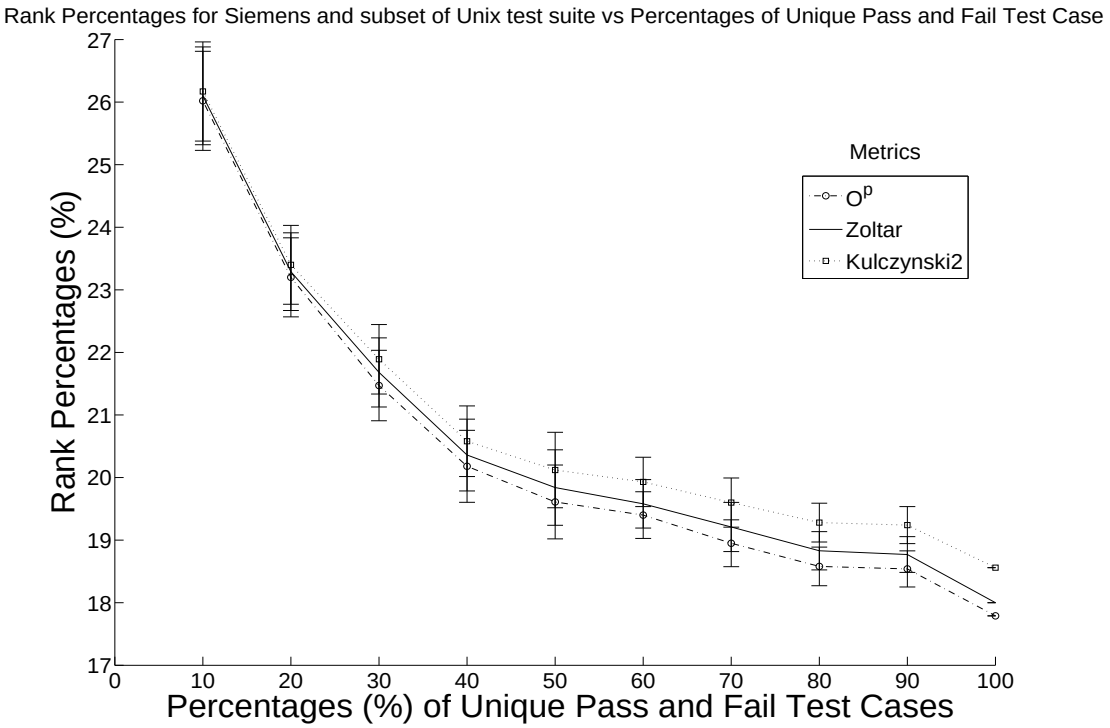

Figure 7.3: Average Rank Percentages (on average) for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite vs Percentages of the Unique Pass and Fail Test Cases

We evaluate similar studies of varying the percentages of unique test cases for the single bug Concordance and Space programs and for the multiple-bug programs of our benchmarks. In these evaluation, we report the p-values for the similar hypothesis that we have established earlier for the single bug Siemens Test Suite and the subset of the Unix Test Suite. These figures can be found in Appendix E. In these evaluation, we observe that the bug localization performance improves as we use a larger number of unique test cases. For Concordance (Table E.1), the improved bug localization performance using all the unique test cases as compared to using 10% of the unique pass and fail test cases on better performing metrics is only significant with confidence of 84%. We observe improvement in bug localization performance by using larger number of unique test cases for the single bug Space programs (Table E.2), and multiple-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite (Table E.3 and Table E.4). In these programs, we observe the statistically significant improvement in bug localization performance using all the unique test cases as compared to using 10% of the unique pass and fail test cases, for all the better performing metrics with confidence greater than 95%. The improved bug localization performance using all the unique test cases as compared to using 10% of the unique pass and fail test cases on the better performing metrics in multiple-bug programs ranging from the average rank percentages of 5.83% to 11.45%.

7.5 Summary

By using unique test cases, we observed most of the benchmarks (Siemens Test Suite, subset of Unix Test Suite, Concordance, and Space) did not show any degradation of the bug localization performance for better performing metrics. We also performed experimental study on varying the percentages of unique test cases with respect to bug localization performance for all the single bug and multiple-bug programs of our benchmarks. From our evaluation, we found improvement in bug localization performance by using all the unique test cases compared to using 10% of the unique pass and fail test cases. The latter improvement is statistically significant with confidence greater than 95% for better performing metrics on most of our benchmarks. The improvement in bug localization performance for the better performing metrics, especially on multiple-bug programs of our benchmarks is in the range of 5.83% to 11.45%.

8

Weighted Incremental Ranking Approaches

8.1 Introduction

In earlier chapters, we use program spectra to locate bugs. We will now propose some improvements based on the *heuristics* from the test cases, especially fail test cases. Initially, we associate varying *weights* with fail test cases — test cases that execute fewer statements are given more weight and have more influence on the ranking. This generally improves diagnosis accuracy with little additional cost. We also propose to rank the program statements incrementally. After the top-ranked statement is identified, the weights are adjusted in order to compute the rest of the ranking. This further improves accuracy. The cost is greater, but not prohibitive. Previous studies [Jones and Harrold, 2005, Abreu et al., 2006] locate bugs without using any weights. We name such an approach as Unweighted.

We introduce an approach where different weights are assigned according to the information of the fail tests. This is followed by proposing the incremental ranking approaches. We analyse our proposed weighted and incremental ranking approaches on the model program of $ITE2_8$ (Figure 5.1). We also detail the empirical results of using our proposed weighted and incremental ranking approaches on the Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space benchmarks.

8.2 Varying Weights for Fail Tests

Fail test cases are important in locating bugs. A typical fail test case that executes statements indicates the likelihood of these statements being buggy. By making use of

this knowledge, we want to improve bug localization performance by assigning different weights to the different fail test cases.

The idea behind varying weights comes from the following observations. First, some fail tests provide more information than other fail tests. Consider the extreme example of two fail tests, where one executes almost every statement and the other executes only one statement. The first test gives us little information whereas the second test allows us to conclude with certainty that the executed statement is buggy. By only using the number of fail tests in which a statement is executed, this information is lost. Second, although ranking metrics are normally applied to natural numbers, they can be defined and used on real numbers. With varying weights for fail tests, the a_{ef} and a_{nf} values we compute can be any real number between zero and the number of fail tests. Of the dozens of metrics proposed, none use integer-specific operations such as modulo; no adaptation is needed for their use with non-integral a_{ef} and a_{nf} values. For the sake of readability, simpler terminology is used for some of the terms that have already been defined in the Glossary. F denotes the total number of fail tests instead of $totF$.

The weights we use depend on the set of *suspect* (possibly buggy) statements, B. Initially, these are the statements executed in at least one fail test. The execution information is represented as a matrix of binary numbers $e_{s,t}$ (as shown in Table 2.2). This is refined in the incremental ranking approach described later. We first define a relative weight, w_t, for each test t, which is (almost) inversely proportional to the number of *suspect* statements executed in the test minus one:

Definition 19 (Relative Weight, w_t)**.**

$$w_t = \frac{1}{\sum_{s \in B} e_{s,t} - 1 + \epsilon}$$

We use a small constant, ϵ (we use 0.0001 in our code), to avoid division by zero when there is only one *suspect* statement executed. The weight given to a fail test t is $w_t F/W$, where W is the sum of the relative weights w_t over all fail tests: $\sum_{t=1}^{F} w_t$. The total weight of all fail tests is thus unaffected by the weighting, so the overall impact of the fail tests (compared to pass tests) is not changed. Also, if all fail tests execute the same number of statements, all weights are 1 (and the method is equivalent to the Unweighted method). To compute a_{ef} for a statement, we take the sum of the weights of fail tests in which it was executed, rather than simply the number of tests.

Definition 20 (Weighted a_{ef})**.**

$$a_{ef} = \sum_{e_{s,t}=1} w_t F/W$$

By using the same example of Table 2.2, if we ignore ϵ, the relative weights for tests 1–3 are 1/4, 1/4, and 1/2, respectively. The a_{ef} values for statements 1–6 are 3, 3, 1.5, 1.5, 2.25, and 0.75, respectively. The a_{nf} values can be computed using the same weighted sum (over the fail tests where the statement is not executed). Alternatively, we can simply use the total number of fail tests minus a_{ef}. In this example, the a_{nf} values are therefore 0, 0, 1.5, 1.5, 0.75, and 2.25, respectively. The greater weight for test 3 leads to the a_{ef} for statement 5 being greater than the a_{ef} for statement 3 and statement 4. For most of the better metrics, this is sufficient to raise the ranking of statement 5 above that of statement 3 and 4.

In general, the weightings result in more rational behaviour than the Unweighted method. If there is one fail test that executes a single statement s, that test will have relative weight of $1/\epsilon$ and a weight close to one, whereas other weights will be close to zero. Thus, the a_{ef} of the statement s will be close to F, while the a_{ef} for other statements will be close to zero. The better ranking metrics will rank s highest (it is as if statement s is used in all the fail tests and no other statement is used in any fail test).

The weighting approach we propose is not just a generic method for determining *similarity* of a row of the matrix of Table 2.2 with the result vector — it uses knowledge of the software diagnosis domain. As this approach involves weighting of fail tests, we name this approach as Weighted throughout this chapter.

Algorithm 30: Algorithm of Weighted Approach

Input: program spectra
Output: ranking (sequence of statements)
foreach *fail test case* **do**
    Determine the number of *suspect* statements in the test;
    Determine the relative weight, w_t, for the test (Definition 19) ;
end
foreach *statement* **do**
    Compute the a_{ij} for the statement including the weighted a_{ef} (Definition 20) ;
    Compute the $MetValue$ of respective spectra metrics;
end
Rank and sort all the statements based on the $MetValue$ in descending order;

Using weights does not change the algorithmic complexity of ranking. We show the Weighted approach in Algorithm 30. S, T, and F refer to the number of statements, test cases, and fail test cases respectively. Initially, the *suspect* and the relative weight w_t is determined for each fail test case of F. This is followed by computing the a_{ij} values (including weighted a_{ef} in Definition 20) and the metric value for each statement s. This gives a time complexity for the Weighted approach of $O(ST)$. The Unweighted approach also has $O(ST)$ if the whole matrix of execution data is stored and each statement s is processed at a time to compute the a_{ij} values. Having computed the a_{ij} values, the metric

values can be determined and then sorted in $O(S \log S)$ time to give the ranking. This gives an overall complexity for the Weighted approach of $O(ST + S \log S)$ time.

In terms of space complexity, if the whole matrix is not stored, the Unweighted approach only takes $O(S)$ to compute the a_{ij} values, instead of $O(ST)$. Accumulators can be used for the a_{ij} values while processing one test at a time. The same space complexity of $O(S)$ applies for the Weighted approach to compute the weighted values. From Algorithm 30, relative weight can be computed for each fail test. An accumulator can be used for the relative weight of each fail test. An accumulator can also used to compute the number of fail tests. After processing the last test, the accumulators for the relative weights of each fail test can be multiplied with the accumulator of the total number of fail tests, divided by W (the sum of all the accumulators of the relative weights of fail tests) to compute the weighted a_{ef} values (Step 6).

8.3 Incremental Ranking Approaches

The model of debugging suggested by ranking statements (by any method) is that the top-ranked statement is considered first and the second-ranked statement is only considered when the programmer has established the top-ranked statement is correct. Although naive, this model is typically assumed when evaluating performance of ranking methods. It also allows us to use additional information to refine the ranking, and generate it incrementally. Once the top-ranked statement has been decided, other statements can be ranked under the *assumption* that the top-ranked statement is *not* buggy. With a probabilistic approach, for example, the statement with the highest probability of being buggy would be ranked highest. The next highest ranking would be the statement with the highest probability of being buggy *given that* the top-ranked statement is correct, and so on.

Generating a ranking incrementally in this way is not a particularly novel technique. However, it cannot refine traditional (Unweighted) spectral ranking because there is no method of incorporating the additional information: the metric value returned for a statement does not depend on information about other statements. In contrast, the Weighted approach introduced depends on the set of suspect statements. By no longer considering the top-ranked statement to be suspect, the number of suspect statements executed in a test decreases and the weights change, potentially changing the ranking. We use pseudo-random numbers to break the ties of the weighted statements and pick the highest ranked statement in each iteration.

For example in Table 2.2, having computed the metric values using the Weighted approach, statement 1 (S1) and statement 2 (S2) are ranked highest with all sensible metrics. Assuming that S1 is ranked slightly higher than S2 (due to the pseudo-random numbers

that we introduce to break the ties), S1 is no longer considered suspect in the next iteration. Therefore, the number of suspect statements executing test T1, T2, and T3 are 4, 4, and 2 respectively. In the next iteration, S2 is ranked highest by all sensible metrics. S2 is no longer considered suspect in the following iteration. T3 has a relative weight of $1/\epsilon$ and statement 5 (S5) (the only remaining suspect statement executed in T3) is ranked highest by all sensible metrics. Note this ranking is different (and more rational) than the ranking produced by the Unweighted method. As this approach involves ranking statements starting with the highest ranked statement, we name this approach *top-down* incremental ranking (Incre.).

Algorithm 31: Algorithm of Top-down Incremental Ranking Approach (Incre.)

Input: spectra coverage including Pass/Fail of the tests
Output: ranking (sequence of statements)
`// Let ranking as empty array`
1 **while** *any fail tests that executes at least one* suspect *statement* **do**
2 Compute fail tests weights using *suspect* and spectra coverage;
3 Compute a_{ij} and metric values for each statement;
4 Determine statement s with maximum metric value;
5 Append s to the *ranking*;
6 Remove s from *suspect* and spectra coverage (assume s is not buggy);
7 **end**
8 Exit;

The pseudocode of the *top-down* incremental ranking approach (Incre.) is described in Algorithm 31. It uses the spectra coverage as inputs and outputs the *ranking* array which consists of a sequence of statements. It terminates when all statements executed in at least one fail test have been ranked. This ensures a buggy statement appears in the ranking, and prevents weights from becoming negative. In our example, it terminates after ranking statement S1, S2, and S5 — one of these must be buggy so it is not necessary to consider the other two statements at all in our search for a bug. When several statements have the maximal metric value, an arbitrary one is chosen (our implementation picks one pseudo-randomly). In order to avoid bias in choosing one particular statement randomly, we evaluate the incremental ranking approach on each program 20 times, and the average of these evaluation is taken. Note that the chosen statement with the maximum metric value might be different, which could affect the ranking of the buggy statement. Therefore, there might be some fluctuation of the bug localization performance using the incremental ranking approach.

The time taken to find the top-ranked statement is $O(ST)$, and the time taken to produce the entire ranking is $O(S^2T)$. This is a substantial increase in overall CPU time. However, it is possible to compute the lower parts of the ranking in parallel with manually checking correctness of the higher ranked statements (though this does constrain the

user interface in a debugging tool). The manual checking is likely to take more time, so the main bottleneck is determining the top-ranked statement, which has acceptable complexity. Space is another additional cost. The weights depend on which statements are *suspect*, and as this changes during execution, the matrix of Table 2.2 (for fail tests at least) is required. If the matrix is stored in the main memory, the space complexity is $O(ST)$. The alternative is re-reading it from the disk in each iteration, reducing space complexity to $O(S)$ but slowing the algorithm by a substantial constant factor. Overall, although the decrease in efficiency is significant, it does not make the algorithm infeasible.

One way to reduce the CPU time is to use the incremental ranking approach to rank only some statements, and rank the remaining statements using the simpler Weighted approach (with the remaining set of suspect statements). In Section 8.5, we provide performance figures for ranking the top 10% and 20% of statements incrementally. These approaches will stop immediately if the programmer recognises the top-ranked statement that is likely the bug. In practice, the bug is not known by the programmer. If the bug is not found after ranking all the 10% and 20% of the top-ranked statements, the algorithm will stop.

The other possible variation of the incremental ranking approach is to generate the ranking *bottom-up* — starting with the lowest ranked statement. Similar weight treatment can be applied for each program statement. The statement with the lowest probability of being buggy (metric value) is ranked lowest. The lowest-ranked statement is often least likely to be the bug. Therefore, all the statements of the program have to be ranked before the complete ranking list of statements (likely to be buggy) is presented to the programmers. This approach takes a longer time and is not practical in debugging. We do not evaluate this approach in this chapter.

The *bottom-up* ranking approach is not a particularly new approach. Guyon et al. applied this approach, which referred to as the support vector machine (SVM) method of Recursive Feature Elimination (SVM-RFE) in their study [Guyon et al., 2002]. This approach is used to narrow down the gene expressions (features of genes) that are responsible for leukaemia and colon cancer. The proposed SVM-RFE uses the SVM classification algorithm [Boser et al., 1992, Vapnik, 1998] to train, classify, and discover features of genes that are responsible for cancer tissues. Initially, the SVM algorithm is applied on the genes dataset to compute the weight for each feature of the genes that are likely to cause the cancer. This is followed by ranking the features of the genes. The gene feature that gives the lowest weight will be eliminated and will not be considered as part of the feature of the genes in the next iteration. A feature rank list containing the feature of the genes with the lowest weight is maintained in each iteration. The SVM algorithm is applied again in the next iteration. The feature of the genes with the next lowest weight

is eliminated and the feature rank list is updated. These steps are repeated until all the features of the genes have been ranked in the feature rank list.

8.4 Using Proposed Weighted and Incremental Ranking Approaches on Model Program

We found that O and O^p are optimal metrics for the single bug programs in Chapter 5. In this section, we want to investigate whether the proposed incremental ranking approaches affect the ranking of these optimal metrics in the single bug programs. We use the $ITE2_8$ model program (Figure 5.1) to evaluate our proposed incremental ranking approaches on these metrics.

Table 8.1: Evaluation of Incremental Ranking Approaches on the Model Program $ITE2_8$

Average Rank Percentages (on average of 100 times)			
Metric	Unweighted	Weighted	Incre.
Number of Tests=2			
O	35.00	35.00	38.10
O^p	35.00	35.00	34.94
Number of Tests=5			
O	31.85	31.85	32.03
O^p	31.85	32.03	32.02
Number of Tests=10			
O	29.72	29.72	29.74
O^p	29.72	30.33	30.33
Number of Tests=100			
O	25.80	25.80	25.80
O^p	25.80	26.26	26.26

Table 8.1 shows bug localization performance of the evaluation of the $ITE2_8$ model program on different number of tests (2, 5, 10, and 100 tests) with 1 million points. To avoid bias on certain single generated multiset paths, we perform the evaluation for 100 times and take the average. On 2 tests, we observe that the bug localization performance for the Unweighted and Weighted approaches on the O and O^p metrics are the same, with the average rank percentages of 35.00%. For the top-down incremental ranking approach (Incre.), we observe a slight improvement of bug localization performance (improved average rank percentages of 0.06%) on the O^p metric as compared to the Unweighted approach. As we increase the number of tests beyond 2, we observe that the effectiveness of bug localization performance for the Weighted approach drops slightly for the O^p metric

as compared to the Unweighted approach. However, the O metric still shows no differences in bug localization performance for both Unweighted and Weighted approaches for all the tests.

The O^p metric (refer to the metric in Table 2.3) ranks statements primarily on a_{ef} and secondarily on a_{ep}. When applying the Weighted approach (Algorithm 30) on the multiset paths of $ITE2_8$ (Figure 5.1), it is possible for $S1$ or $S2$ (the non-buggy statements) to have an a_{ef} value close to the a_{ef} value of $S4$ (the buggy statement). At the same time, $S1$ or $S2$ can possibly has lower a_{ep} value than the a_{ep} value of $S4$. Using the O^p metric, $S1$ or $S2$ can potentially ranks higher than the buggy statement in some cases.

Even though the weighted $S1$ or $S2$ has an a_{ef} value close to the a_{ef} value of $S4$, these statements are not executed by *all* the fail test cases. $S1$ or $S2$ has the property that $a_{nf} > 0$. For the O metric, these statements will have very small metric values (refer to the O metric in Table 2.3). Therefore, the cases of $S1$ or $S2$ ranking higher than the buggy statement $S4$ are not observed using O metric, and we do not observe any drop in the bug localization performance for the Weighted approach for this metric in Table 8.1.

In Table 8.1, the top-down incremental ranking approach (Incre.) does not show consistent bug localization improvement on the optimal metrics O and O^p. In the $ITE2_8$ model program, either $S1$ or $S2$ and the bug $S4$ are executed by fail test cases. Using pseudo-random numbers to break the ties of choosing the highest-ranked statement might introduce some *noise* and affect the bug localization performance of the incremental ranking approach. We already attempt to reduce the *noise* by repeating the evaluation 20 times and taking the average of the evaluation.

Generally, we do not observe any improvement in bug localization performance by using our proposed Weighted and top-down incremental ranking approach (Incre.) on the optimal metrics O and O^p of the $ITE2_8$ model program (single bug program). There might be different observations of bug localization performance for O and O^p metrics on multiple-bug programs. For a typical multiple-bug program, the buggy statements are not necessarily executed by all the fail test cases. These statements might have different weights assigned using the Weighted and the incremental ranking approaches. We evaluate our proposed approaches empirically in Section 8.5.

8.5 Empirical Evaluation of the Proposed Weighted and Incremental Ranking Approaches

In this section, we evaluate our proposed Weighted and incremental ranking approaches on our benchmarks, namely the Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space programs. For each dataset, we compare the Unweighted approach with the Weighted and incremental ranking approaches. In the incremental ranking approaches, we rank the top 10% (10% Inc) and 20% (20% Inc) of the statements. We also rank all the statements using the top-down incremental ranking approach (Incre.). In the 10% Inc, 20% Inc, and Incre. approaches, it is possible to have more than one highest ranked statement. We use pseudo-random numbers to break ties of the statements and pick the highest ranked statement. The bug localization performance might vary as these statements can have different random orderings. We attempt to reduce this *noise* by repeating each evaluation 20 times. We report the average of the bug localization performance of the evaluation.

For Space programs, we evaluate all the 10 bins which consist of the different *Subset* of the entire test suite of Space (refer to the details in Section 4.5). We take the average of the bug localization performance for all the 10 bins in the evaluation.

8.5.1 Single Bug Programs

Table 8.2: Average Rank Percentages for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	17.86	17.86	18.01	18.04	18.00	0.5066
O^p	17.86	17.87	17.87	17.81	17.83	0.3997
Wong3	18.19	18.06	18.08	18.08	18.02	0.1727
Zoltar	18.23	18.22	18.13	18.16	18.16	0.6014
Wong4	18.93	18.76	19.24	19.15	19.03	0.9976
Kulczynski2	19.06	18.91	18.98	18.97	18.98	0.2109
McCon	19.06	18.91	18.80	19.04	19.00	0.6154
JacCube	20.06	19.58	19.34	19.37	19.55	0.0945
M2	20.12	19.73	19.82	19.74	19.60	0.0468
Ochiai	21.63	21.18	20.95	20.55	20.87	0.0105
Pearson	23.56	23.27	23.17	23.19	23.31	0.0472
Jaccard	23.64	23.09	23.19	23.25	23.27	0.0051

Continued on next page

Table 8.2 – continued from previous page

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
AMean	23.66	23.19	23.27	23.33	23.43	0.1657
Ample2	24.08	23.65	23.68	23.67	23.72	0.0157
Rogot2	24.88	24.11	24.01	24.08	24.15	0.0618
CBI Log	26.80	25.61	25.70	26.04	26.02	0.0336
Tarantula	27.09	26.49	27.35	27.34	27.36	0.4311
Binary	30.02	30.02	29.82	30.02	29.54	0.4349
Russell	30.02	30.02	30.01	30.55	29.92	0.4748
Ample	30.16	27.39	27.43	27.22	27.38	0.0012
Overlap	32.23	32.20	27.93	29.07	32.95	0.5298

Table 8.3: Average Rank Percentages for the Single Bug Concordance Programs

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	10.11	10.11	10.05	10.06	10.03	0.3631
O^p	10.11	10.11	10.07	10.23	10.04	0.2875
Zoltar	10.11	10.11	10.08	10.11	10.09	0.2875
Wong3	10.15	10.11	10.02	10.09	10.00	0.0379
Kulczynski2	10.21	10.35	10.35	10.29	10.35	0.3120
McCon	10.21	10.35	10.38	10.30	10.33	0.6822
JacCube	10.43	10.56	10.60	10.62	10.56	0.1568
M2	10.97	10.61	10.58	10.65	10.66	0.5472
Ochiai	11.19	10.76	10.73	10.69	10.79	0.7232
Wong4	11.35	11.43	20.12	17.30	16.76	1
Jaccard	17.68	11.43	11.37	11.30	11.38	0.2641
Ample2	18.05	11.00	11.07	11.07	11.05	0.4528
Pearson	18.42	11.17	11.24	11.25	11.14	0.3999
AMean	18.90	11.56	11.42	11.59	11.44	0.0914
Rogot2	19.24	18.70	18.76	18.75	18.76	0.2771
Tarantula	20.03	12.78	12.55	12.59	12.71	0.1869
Binary	21.03	21.03	21.85	18.87	21.67	0.5294
Overlap	21.03	21.03	17.02	20.28	20.86	0.6499
Russell	21.03	21.03	20.73	20.47	21.76	0.8205
CBI Log	22.63	12.46	12.44	12.87	12.63	0.4192

Continued on next page

Table 8.3 – continued from previous page

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Ample	27.53	24.02	23.80	22.17	22.03	0.1429

Table 8.4: Average Rank Percentages (on average of 10 bins) for the Single Bug Space Programs

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	1.64	1.64	1.71	1.70	1.73	0.3193
O^p	1.64	1.64	1.64	1.65	1.62	0.6334
Wong4	1.64	1.70	1.70	1.71	1.71	0.4333
Wong3	1.65	1.64	1.64	1.61	1.64	0.0191
Zoltar	1.80	1.81	1.82	1.80	1.81	0.4435
JacCube	1.90	1.90	1.91	1.91	1.91	0.3649
M2	1.92	1.93	1.92	1.93	1.92	0.6441
Kulczynski2	2.07	2.08	2.10	2.09	2.12	0.9794
McCon	2.07	2.08	2.10	2.09	2.07	0.4452
Ochiai	2.26	2.32	2.34	2.35	2.34	0.1262
Rogot2	2.67	2.65	2.63	2.62	2.64	0.9822
Ample2	2.68	2.60	2.60	2.59	2.59	0.7612
Pearson	2.72	2.71	2.70	2.71	2.70	0.2997
AMean	2.93	2.90	2.89	2.91	2.91	0.3124
Jaccard	3.18	3.10	3.11	3.08	3.10	0.7744
Tarantula	6.31	6.02	6.09	6.12	6.09	0.1651
Ample	6.56	3.99	3.80	3.80	3.78	0.0151
CBI Log	6.65	6.15	6.35	6.25	6.24	0.9681
Binary	17.59	17.59	17.91	18.06	18.21	0.8634
Russell	17.59	17.59	17.42	17.68	17.62	0.598
Overlap	18.31	18.28	16.39	18.43	18.12	0.2106

For our evaluation on all the single bug programs (Table 8.2, Table 8.3, and Table 8.4), we observe that the O metric does not show any difference in bug localization performance using the Weighted approach as compared to using the Unweighted approach. This concurs with the observation of the O metric using the $ITE2_8$ model program in Section 8.4. O^p metric shows no difference in bug localization performance using the

Weighted approach as compared to using the Unweighted approach in all the single bug programs except the Siemens Test Suite and the subset of the Unix Test Suite. There is a slight drop in the effectiveness of bug localization performance on this metric using the Weighted approach as compared to the Unweighted approach in Table 8.2 (17.87% and 17.86% respectively). The drop of the effectiveness of bug localization performance on this metric is similarly observed using the model program $ITE2_8$ in Section 8.4. In Table 8.2, Table 8.3, and Table 8.4, metrics which are at the bottom of the tables (e.g AMean and Tarantula to name a few) show that the bug localization performance using the Weighted approach outperforms the Unweighted approach.

The O and O^p metrics only show marginal improvement of bug localization performance for the top 10% (10% Inc) approach as compared to the Unweighted approach in the single bug Concordance programs, with improved average rank percentages of 0.06% and 0.04% respectively. We observe a slight fluctuation of the bug localization performance on the O^p metric, where the bug localization performance improves using the top 20% (20% Inc) approach but not, using the top 10% (10% Inc) approach on the single bug Siemens Test Suite and the subset of the Unix Test Suite. Other metrics that are not at the top of the tables of our evaluation of single bug programs, namely Overlap, also shows fluctuation in bug localization performance using the top 10% (10% Inc) and 20% (20% Inc) approaches. This is due to the way we use pseudo-random numbers to break ties in choosing the highest-ranked statement. Statements that are supposedly ranked just below the buggy statement using the Unweighted approach can be ranked before the buggy statement in the 10% Inc and 20% Inc approaches. Therefore, we observe a slight fluctuation in bug localization performance for some of these metrics.

For the top-down incremental ranking approach (Incre.), only the O^p and Wong3 metrics show improved bug localization performance compared with using the Unweighted approach for all the single bug programs. The latter improvement of average rank percentages is in the range of 0.01% to 0.17%.

In the evaluation of all the single bug programs (Table 8.2–Table 8.4), it is not clear how significant is the improvement in bug localization performance using our proposed top-down incremental ranking approach (Incre.) as compared to the Unweighted approach. Therefore, we report the p-value [Rice, 1989] of our hypothesis in these tables. We establish the hypothesis that *the bug localization performance using our proposed top-down incremental ranking approach (Incre.) improves as compared to using the Unweighted approach*. One-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] is applied to check the statistical significance of our hypothesis. We perform this statistical test on one of the 10 bins in the Space programs as the bug localization performance across the 10 bins are very similar (see Figure 5.13 of Subsection 5.8.1).

Generally the bug localization performance using the top-down incremental ranking approach (Incre.) as compared to the Unweighted approach gives only weak support for our hypothesis in the single bug programs. We can say with reasonable confidence that performance for top metrics such as O^p metric does not significantly degrade in this case. Stronger statistical evidence requires a larger set of benchmarks.

8.5.2 Multiple-bug Programs

We evaluate our proposed approaches on the multiple-bug Siemens Test Suite, subset of the Unix Test Suite, and Space programs (Table 8.5, Table 8.6, and Table 8.7). We also evaluate on the *Subset* of the test suite of the Space for multiple-bug Space programs. We report the average of the bug localization performance for the 10 bins of Space programs in Table 8.7.

Table 8.5: Average Rank Percentages for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Kulczynski2	19.53	19.35	18.05	17.83	18.00	<0.05
McCon	19.53	19.35	17.98	18.03	17.91	<0.05
Ochiai	20.18	19.62	18.22	18.22	18.17	<0.05
JacCube	20.51	20.27	19.07	19.01	18.99	<0.05
Zoltar	20.52	20.46	19.11	19.18	19.17	<0.05
Wong4	20.78	20.46	19.76	19.43	19.40	<0.05
M2	20.80	20.70	19.31	19.38	19.24	<0.05
AMean	20.97	20.29	19.08	18.98	18.98	<0.05
Pearson	21.09	20.26	18.97	18.92	18.91	<0.05
Jaccard	21.10	20.38	19.09	19.06	19.03	<0.05
Ample2	21.37	20.59	19.32	19.30	19.28	<0.05
Rogot2	21.78	21.19	19.70	19.72	19.75	<0.05
CBI Log	22.29	21.50	20.44	20.46	20.47	<0.05
Wong3	22.54	22.44	21.05	21.07	21.14	<0.05
O^p	22.84	22.72	21.40	21.41	21.28	<0.05
Tarantula	23.13	22.27	21.32	21.23	21.28	<0.05
Ample	24.49	24.17	23.11	23.02	23.03	<0.05
O	24.95	24.66	22.65	22.52	22.67	<0.05
Russell	32.10	32.01	26.79	26.60	26.65	<0.05
Binary	34.02	33.75	27.57	27.74	27.78	<0.05

Continued on next page

Table 8.5 – continued from previous page

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Overlap	34.02	34.02	23.87	24.90	26.99	<0.05

Table 8.6: Average Rank Percentages for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Kulczynski2	21.94	21.61	17.59	17.57	17.54	<0.05
McCon	21.94	21.61	17.60	17.49	17.62	<0.05
AMean	22.24	21.84	17.92	17.84	17.77	<0.05
Pearson	22.31	21.83	17.92	17.90	17.86	<0.05
Rogot2	22.53	22.14	17.97	18.06	18.04	<0.05
Ample2	22.56	22.11	18.20	18.16	18.14	<0.05
Ochiai	22.60	22.25	18.22	18.20	18.19	<0.05
Wong4	22.60	21.62	18.43	18.17	17.97	<0.05
Jaccard	23.12	22.69	18.63	18.54	18.62	<0.05
CBI Log	23.37	23.11	19.32	19.38	19.33	<0.05
Zoltar	23.49	23.44	19.32	19.40	19.35	<0.05
Ample	23.88	23.54	19.60	19.60	19.52	<0.05
JacCube	24.86	24.63	20.54	20.52	20.54	<0.05
M2	24.97	24.88	20.74	20.75	20.69	<0.05
Wong3	25.24	25.18	21.04	20.96	21.05	<0.05
O^p	25.33	25.25	21.05	21.02	21.08	<0.05
Tarantula	27.23	26.26	20.58	20.48	20.44	<0.05
Russell	28.67	28.65	23.06	23.03	22.97	<0.05
O	29.29	28.90	22.84	22.74	22.66	<0.05
Binary	32.43	32.05	24.39	24.44	24.37	<0.05
Overlap	38.34	38.42	23.65	23.96	24.43	<0.05

Table 8.7: Average Rank Percentages (on average of 10 bins) for the Multiple-bug Space Programs

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Zoltar	2.67	2.67	2.27	2.27	2.24	<0.05
JacCube	2.75	2.74	2.31	2.33	2.34	<0.05
O	2.80	2.80	2.40	2.38	2.37	<0.05
O^p	2.80	2.80	2.37	2.39	2.39	<0.05
Wong3	2.80	2.80	2.40	2.39	2.39	<0.05
M2	2.83	2.83	2.41	2.39	2.41	<0.05
Kulczynski2	2.91	2.91	2.52	2.50	2.50	<0.05
McCon	2.91	2.91	2.51	2.49	2.50	<0.05
Wong4	3.09	3.03	2.61	2.60	2.63	<0.05
Ochiai	3.13	3.10	2.70	2.71	2.71	<0.05
Jaccard	3.70	3.66	3.25	3.24	3.23	<0.05
Rogot2	3.75	3.71	3.27	3.28	3.28	<0.05
Ample2	3.84	3.79	3.35	3.35	3.36	<0.05
Pearson	3.84	3.76	3.32	3.32	3.31	<0.05
AMean	4.00	3.95	3.47	3.47	3.48	<0.05
CBI Log	4.21	4.19	3.78	3.77	3.77	<0.05
Tarantula	4.85	4.79	4.37	4.38	4.38	<0.05
Ample	8.82	4.83	4.42	4.40	4.41	<0.05
Binary	19.62	19.62	11.54	11.57	11.68	<0.05
Russell	19.62	19.62	11.48	11.70	11.80	<0.05
Overlap	19.87	19.86	11.30	11.39	11.31	<0.05

In Table 8.5 and Table 8.6, using the top-down incremental ranking approach (Incre.) helps to improve bug localization performance as compared to using the Unweighted approach. The differences of the bug localization performance between the top-down incremental ranking approach (Incre.) and the Unweighted approach are more obvious in Table 8.6 as compared to Table 8.5. Better performing metrics such as Kulczynski2 and McCon show that the improved average rank percentages are 0.18% and 0.33% in Table 8.5 and 8.6 using the Weighted approach as compared to the Unweighted approach. We also observe improvement in the bug localization performance for all the metrics in these tables using the 10% Inc and 20% Inc approaches as compared to the Unweighted approach. In the 10% Inc and 20% Inc approaches, the improvement of bug localization

performance for all the metrics in these tables ranges from the improved average rank percentages of 0.39% to 14.69% as compared to the Unweighted approach.

For the multiple-bug Space programs (Table 8.7), we observe that most metrics do not show much differences in bug localization performance using the Weighted approach as compared to the Unweighted approach. Generally, all the metrics in this table observe improvement in bug localization performance using the incremental ranking approaches (10% Inc, 20% Inc, and Incre.) as compared to the Unweighted approach.

We also report the p-value of our hypothesis (as defined in Subsection 8.5.1) in all the multiple-bug programs (Table 8.5, Table 8.6, and Table 8.7). The p-value of our hypothesis for all the metrics in the multiple-bug programs is less than 0.05. Therefore, the improved bug localization performance using our proposed top-down incremental ranking approach (Incre.) as compared to the Unweighted approach is statistically significant with confidence greater than 95% for all the metrics. For the multiple-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite (Table 8.5 and Table 8.6), we also evaluate the p-value to compare the improvement of bug localization performance using our proposed Weighted approach and the Unweighted approach. In these tables, all the better performing metrics show statistically significant improvement in bug localization performance using the Weighted approach as compared to the Unweighted approach, with confidence greater than 95%. The latter improvement ranges in the average rank percentages of 0.06% to 0.98%.

8.5.3 Time Taken

We report the average of the time taken to locate program bugs using the Unweighted, Weighted, and incremental ranking approaches. Note that our implementation of the proposed Weighted and the incremental ranking approaches are *naive* and does not consider any optimisation of the algorithm. Table 8.8 reports the time taken to compute and locate bugs of the Siemens Test Suite and subset of the Unix Test Suite. The time taken to compute and locate the single bug of Concordance and Space programs is shown in Table 8.9. We also show the time taken to compute and locate the multiple-bug Space programs in Table 8.10.

The time taken to evaluate the Siemens Test Suite, the subset of the Unix Test Suite, and Concordance programs (Table 8.8 and Table 8.9) increases uniformly by using the Unweighted, Weighted, 10% Inc, 20% Inc, and Incre. approaches. The time taken to compute the weights and produce the ranking of statements for the Weighted approach is slightly more than the Unweighted approach. For the 10% Inc and 20% Inc approaches, a longer time is needed to compute the weights incrementally and find the next top-ranked statement(s) until the bug is found. For different iterations, weights have to be recomputed

Table 8.8: Time Taken for the Siemens Test Suite and the subset of the Unix Test Suite Programs (on average of one program)

Approach	Siemens Test Suite and the subset of the Unix Test Suite		
	One-bug (sec)	Two-bug (sec)	Three-bug (sec)
Unweighted	0.33	0.10	0.04
Weighted	0.47	0.19	0.20
10% Inc	1.57	0.82	1.46
20% Inc	1.94	1.05	1.85
Incre.	2.06	1.07	1.94

Table 8.9: Time Taken for the Single Bug Concordance and Space Programs (on average of one program)

Approach	Concordance (sec)	Space (sec)
Unweighted	0.10	0.49
Weighted	0.19	3.41
10% Inc	0.82	269.99
20% Inc	1.05	286.87
Incre.	1.07	289.78

Table 8.10: Time Taken for the Multiple-bug Space Programs (on average of one program)

Approach	Space (sec)
Unweighted	0.58
Weighted	1.92
10% Inc	114.99
20% Inc	132.21
Incre.	148.66

each time to find the next top-ranked statements. In our implementation of the 10% Inc and 20% Inc approaches, the algorithm stops once the bug is recognised by the programmer. The time increases for the top-down incremental ranking approach (Incre.) as more iterations are needed before the bug is found (Algorithm 31).

There is a huge increase of time taken to locate bugs of the program using the 10% Inc, 20% Inc, and Incre. approaches for single bug and multiple-bug Space programs in Table 8.9 and 8.10. The size of the Space programs is larger than the Siemens Test Suite and the subset of the Unix Test Suite, and Concordance benchmarks. Therefore, we expect the time taken to compute the weights for all the statements and to find the next top-ranked statements (10% Inc, 20% Inc, and Incre.) in Space programs to be longer.

8.6 Summary

In this chapter, we proposed two enhancements to the traditional spectral ranking method for diagnosing software errors. The first is to have variable weights for different fail tests, dependent on the number of statements executed in the tests. The second is to compute the ranking incrementally, and relies on the first method. Our proposal for varying weights of fail tests considers domain-specific knowledge in order to maximise the bug localization performance. We used statistical tests to validate our hypothesis of improved bug localization performance using our proposed top-down incremental ranking approach (Incre.) as compared to the Unweighted approach.

For single bug programs, we did not observe any difference in bug localization performance using our proposed Weighted approach as compared to the Unweighted approach on better performing metrics. For the multiple-bug Siemens Test Suite and the subset of the Unix Test Suite programs, we observed a statistically significant improvement in bug localization performance on all better performing metrics using the Weighted approach as compared to the Unweighted approach. The latter improvement is statistically significant with confidence greater than 95% and ranging in the average rank percentages of 0.06% to 0.98%. For all of the multiple-bug programs of our benchmarks, we also showed improvement in bug localization performance using our proposed top-down incremental ranking approach (Incre.) as compared to the Unweighted approach on all the better performing metrics. The latter improvement observed on the better performing metrics is in the range of 0.41% to 4.63% and is statistically significant with confidence greater than 95%.

The time taken for the top-down incremental ranking approach (Incre.) might scale up as the program size increases. This concern could be addressed by adapting the algorithm of our proposed approach to be executed in parallel. We showed the robustness of our proposed Weighted and incremental ranking approaches on unique (non-redundant) test cases of our benchmarks where the bug localization performance improves as compared to using the Unweighted approach (refer to the empirical results in Appendix F). We considered this to be a firm validation of our proposed approaches.

9

Using Spectral Frequency Weighting Function in Bug Localization

9.1 Introduction

We have used mainly binary information of test coverage to locate bugs on several spectra metrics. This refers to using the information of each program statement being executed and not executed by a particular test; indicated by 1 and 0 respectively. We use the terms of *binary information of test coverage* and *traditional binary weighting approach* interchangeably. In this chapter, we propose using more information from test coverage, particularly frequency counts of test execution. This information has been used in SOBER [Liu et al., 2005]. Frequency execution counts of a program statement refers to the number of times the statement is being executed by a particular test. By exploiting more information from the test coverage (frequency execution counts), we want to investigate whether it could help to improve the bug localization performance.

We introduce an approach of using the frequency counts of test executions to map into respective spectra properties a_{np}, a_{nf}, a_{ep}, and a_{ef}. Using the new a_{ij} values, we evaluate several well-known spectra metrics on the Siemens Test Suite and the subset of the Unix Test Suite, Concordance, and Space benchmarks. We conduct statistical tests on the improvement of the bug localization performance using our proposed approach compared to using the binary information of test coverage.

9.2 Introduction to Spectral Frequency Weighting Function

We detailed a typical test coverage in Table 2.1 of Chapter 2. For convenience, we reproduce Table 9.1 and Table 9.2. Table 9.1 refers to a matrix of test executions with each row

Table 9.1: Example of Test Coverage Information (frequency counts) with Tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5
S_1	60	2	100	0	38
S_2	70	65	90	0	45
S_3	25	35	0	4	0
S_4	80	30	0	42	0
S_5	42	0	37	48	81
S_6	0	59	0	0	17
Test Result	Fail	Fail	Fail	Pass	Pass

Table 9.2: Example of Test Coverage Information (binary) and Program Spectra with Tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	0	1	1	0	1	3
S_2	1	1	1	0	1	1	0	1	3
S_3	1	1	0	1	0	1	1	1	2
S_4	1	1	0	1	0	1	1	1	2
S_5	1	0	1	1	1	0	1	2	2
S_6	0	1	0	0	1	1	2	1	1
				⋮					
Test Result	Fail	Fail	Fail	Pass	Pass				

and column for program statement, s and test case, t respectively. This table indicates the number of times a statement is executed by a particular test case. This type of coverage is known as frequency counts coverage. The larger frequency counts of a particular statement of the test case indicates the statement gets executed more frequently by the test case. Additionally, there is a vector (last row) indicating the result; Pass and Fail of each test case. Table 9.2 shows the binary information of test coverage that only consists of 0 and 1, which we used in previous chapters to locate bugs of the program.

Frequency information is important as it indicates how often a particular statement is executed by a corresponding test case. In order to exploit this extra information, the frequencies are mapped using a frequency weighting function of a typical sigmoid function in the range of [0, 1]. Sigmoid functions are also known as sigmoid curves or logistic curves, and have been used in a range of fields including neural network, biology, and economics [Heaton, 2008]. The sigmoid function is defined as the following.

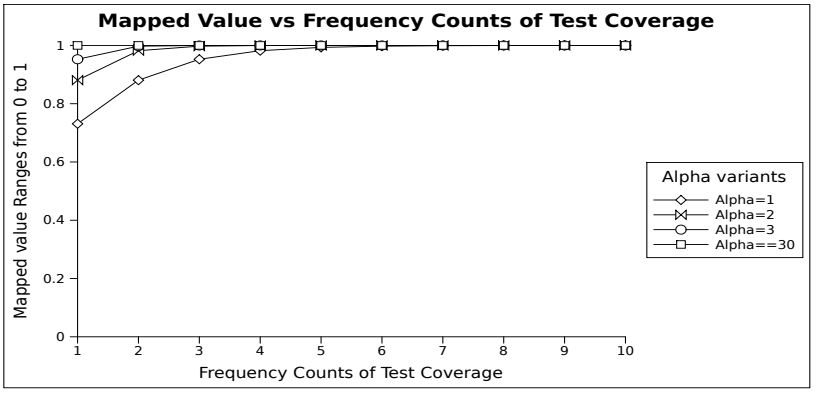

Figure 9.1: Mapped Value vs Frequency Counts

Definition 21 (Sigmoid Function, M)**.**

$$M(k_{st}) = \begin{cases} \frac{1}{1+e^{-\alpha * k_{st}}}, & \text{if } k_{st} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where k is frequency counts of statement, s of test, t and α is a constant

In this definition, the α value refers to a parameter of the sigmoid function, M. We use the sigmoid function M to map the non-zero frequency counts. When there is no execution (frequency count is zero), the function M returns as 0. This function is referred to as the spectral frequency weighting function because it is used to map the respective program spectra properties. Different α values are applied to represent different frequency weighting functions. For illustrative purposes, Figure 9.1 shows a simple mapping of the frequency counts ranging from 1 to 10 using the α value in the range of [1-30]. When the frequency count is 0, the value mapped using the function M for all the α values will be 0. As the mapped value is 0 for all α values, we do not show this point in the figure. As the α value increases, the frequency weighting function curve is steeper. As the α value gets very large (in this example, α value of 30), the mapping approaches to the binary information of test coverage (step function of 0 and 1).

We show a motivated example on the differences between our proposed frequency weighting function approach and the traditional binary weighting approach (binary information of test coverage). Table 9.3 (identical to Table 9.1) is the matrix of test execution information represented in the form of frequency execution counts. It contains the total pass tests, $totP$ of 2 and the total fail tests, $totF$ of 3. We observe Statement 1 (S_1) and Statement 2 (S_2) have been executed by test T_1 for 60 and 70 times respectively. However, S_1 and S_2 have been executed by test T_2 for 2 and 65 times respectively. Using our proposed approach of function M in Definition 21 (assuming α value is 1), the value mapped to both S_1T_2 and S_2T_2 are different (approximately 0.88 and 1.00 respectively). S_1 has

Table 9.3: Example of Program Spectra with Frequency Information and Mapped Program Spectra Properties

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	60	2	100	0	38	1.00	0.12	1.00	2.88
S_2	70	65	90	0	45	1.00	0.00	1.00	3.00
S_3	25	35	0	4	0	1.02	1.00	0.98	2.00
S_4	80	30	0	42	0	1.00	1.00	1.00	2.00
S_5	42	0	37	48	81	0.00	1.00	2.00	2.00
S_6	0	59	0	0	17	1.01	2.00	0.99	1.00
				⋮					
Test Result	Fail	Fail	Fail	Pass	Pass				

been executed by test T_1 and T_3 for 60 and 100 times respectively. The value mapped to both S_1T_1 and S_1T_3 is 1. S_2 has been executed by test T_1 and T_3 for 70 and 90 times respectively. Both S_2T_1 and S_2T_3 is 1. Therefore, a_{ef} for S_1 and S_2 are approximately 2.88 and 3.00 respectively. The a_{nf} and a_{np} for the statements is by taking the subtraction of the a_{ef} and a_{ep} from the $totF$ and $totP$ respectively.

Another example is S_3 and S_4 that have been executed differently by test T_4 (4 and 42 times respectively). S_3T_4 and S_4T_4 would be mapped to approximately 0.98 and 1.00 respectively. Therefore, a_{ep} for Statement 3 (S_3) will be 0.98 and Statement 4 (S_4) will be 1.00.

If we observe Table 9.2 (test coverage information in binary form), S_1 and S_2 are assigned the same metric values since they have identical information of a_{np}, a_{nf}, a_{ep}, and a_{ef}. By using our proposed approach in Table 9.3, the frequency counts information is able to further differentiate the likelihood of a particular statement being buggy. In Table 9.3, we can differentiate that S_2 is more likely to be buggy as compared to S_1. This is due to more fail test cases executing S_2 than S_1 (a_{ef} of S_2 and S_1 is 3.00 and 2.88 respectively).

The frequency weighting function is applied on all the test cases. We explore different α variants of the M function to observe whether changing the α values (different frequency weighting functions) gives any obvious pattern in terms of the improvement of bug localization performance. We expect the values of a_{np}, a_{nf}, a_{ep}, and a_{ef} to change and influence the ranking order of the buggy statements. Different ranking orders of buggy statements indicate the amount of the program code that needs to be examined by the programmer to locate the bugs. Therefore, the performance of bug localization might vary for the different α variants applied to the function M.

9.3 Bug Localization Performance Using Spectral Frequency Weighting Function

We now discuss experiments performed to evaluate the effectiveness of our proposed approach by comparing our proposed approach (using frequency weighting function) with the traditional binary weighting approach. We evaluate these approaches on the Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space benchmarks. We evaluate on the *Subset* of Space programs of the existing Space test cases (in 10 bins). We take the average of the bug localization performance for the 10 bins to reduce any bias of the bug localization performance of any specific bin. The figures for the traditional binary weighting approach on our benchmarks are identical to the figures in which we have reported in Chapter 5.

9.3.1 Single Bug Programs

Initially, we perform our evaluation on the single bug Siemens Test Suite and the subset of the Unix Test Suite programs with respect to the different α values in Table 9.4. The last column (Bin) refers to the traditional binary weighting approach. Other columns refer to our proposed frequency weighting function applied to the different α values (Definition 21). We apply the α values ranging from 0.1 to 100 but only show several of them due to limited space. For example, column 1 refers to applying Definition 21 with the α value being 0.1, column 2 refers to applying the α value being 0.5, and etc.

We observe interesting patterns in bug localization performance using our proposed frequency weighting function on the different α values for several spectra metrics. For readability purposes, we have bolded the figures that give the best bug localization performance among the different α values for several spectra metrics.

All the metrics in the table show the bug localization performance on the different α values converge to the bug localization performance of the traditional binary weighting approach. For best performing spectra metric such as O^p, the performance of bug localization is the best (17.00%) using our proposed approach with the α value 10. The programmer only has to examine 17.00% of the program code in order to locate bugs. By using the traditional binary weighting approach for the same metric, the programmer needs to examine 17.86% of the program code to locate bugs. Some of the metrics show some fluctuation of bug localization performance across the different α values. For example, JacCube metric shows fluctuation of bug localization performance between the α values of 1 and 20.

Tarantula metric shows the best improvement in bug localization performance by us-

ing our proposed approach with the α value of 0.1 as compared to using the traditional binary weighting approach (with 25.81% and 27.09% respectively). Using our proposed approach on any α values, the O metric does not show any improvement in bug localization performance. O metric has the non continuous function property (see the metric in Table 2.3). If a statement of the program has $a_{nf} > 0$, O metric assigns the statement with a very small metric value. Using our proposed approach of frequency weighting function, the buggy statement can have a small a_{nf} value. Therefore, the buggy statement is ranked together with other non-buggy statement(s) (tie exists) and bug localization performance is affected.

To see the trend of bug localization performance on the different choices of α values, we plot the bug localization performance (average rank percentages) against the different α values (ranges from 0.1 to 100) including the traditional binary weighting approach (Bin) for several metrics in Figure 9.2. The metrics shown are O^p, Tarantula, Wong3, Wong4, and Zoltar. It shows that the trend of bug localization performance improves as the α value is between 10 to 20 for the better performing metrics. As the α value gets larger beyond 20, the average rank percentages for all these metrics converge to the average rank percentages using the traditional binary weighting approach (Bin).

Using our proposed approach with the α values ranging between 10 to 50, it has already shown the improvement of bug localization performance for all the metrics in Figure 9.2 as compared to using the traditional binary weighting approach (Bin).

Table 9.4: Average Rank Percentages for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
O	50.46	49.69	48.22	47.38	45.39	44.70	44.13	38.15	**17.86**
O^p	28.68	27.27	26.22	23.83	20.28	17.02	**17.00**	17.21	17.86
Wong3	28.17	26.96	24.52	22.27	18.98	17.79	**17.36**	17.55	18.19
Zoltar	24.16	24.21	24.12	23.99	22.90	18.45	17.71	**17.51**	18.23
Wong4	28.45	27.28	25.65	23.80	20.52	**18.60**	18.75	18.61	18.93
Kulczynski2	24.53	22.33	21.90	19.95	18.47	**18.16**	18.29	18.34	19.06
McCon	24.53	22.33	21.90	19.97	18.48	**18.15**	18.34	18.38	19.06
JacCube	23.06	20.34	19.51	19.63	**19.14**	19.36	19.33	19.53	20.06
M2	24.30	20.88	19.70	19.80	**19.29**	19.33	19.31	19.40	20.12
Ochiai	21.56	20.70	**20.63**	20.77	20.83	20.83	20.87	20.94	21.63
Pearson	22.99	21.98	**21.79**	22.13	22.49	22.61	22.62	22.80	23.56

Continued on next page

Table 9.4 – continued from previous page

Metric	0.1	0.5	1	2	4	8	10	20	Bin
Jaccard	23.30	22.42	**22.37**	22.59	22.73	22.65	22.70	22.91	23.64
AMean	22.76	**21.97**	**21.97**	22.35	22.59	22.75	22.75	23.01	23.66
Ample2	23.38	**22.00**	22.16	22.66	23.02	22.94	22.97	23.25	24.08
Rogot2	23.09	**21.78**	21.79	21.90	22.19	22.24	22.25	23.22	24.88
CBI Log	25.81	25.34	25.35	26.00	26.24	**25.33**	**25.33**	25.98	26.80
Tarantula	**25.81**	25.84	26.02	26.63	26.83	26.66	26.77	26.99	27.09
Russell	31.18	30.35	29.80	29.60	29.54	29.40	28.85	**28.68**	30.02
Binary	50.46	49.75	48.30	47.58	45.98	45.50	45.12	40.45	**30.02**
Ample	28.04	**27.52**	27.94	28.53	28.86	28.75	28.77	29.04	30.16
Overlap	31.19	31.21	**30.87**	31.61	32.01	31.46	31.76	33.27	32.23

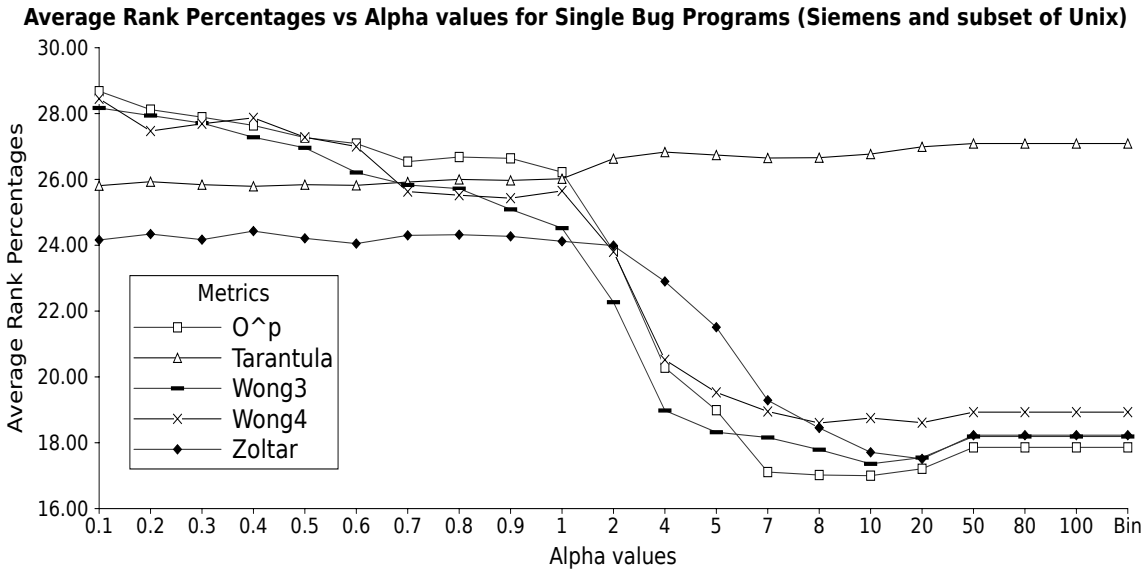

Figure 9.2: Average Rank Percentages for the Different α values of the Single Bug Siemens Test Suite and the subset of the Unix Test Suite

We also evaluate our proposed approach on the single bug programs of Concordance and Space. Similar improvement of bug localization performance is observed using our proposed approach on different α values. The details can be found in Appendix G.

9.3.2 Multiple-bug Programs

In this section, we evaluate our proposed approaches on the multiple-bug programs of the Siemens Test Suite, the subset of the Unix Test Suite, and Space.

Table 9.5: Average Rank Percentages for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
Kulczynski2	20.36	19.00	18.53	18.18	**17.88**	17.97	18.01	18.17	19.53
McCon	20.36	19.00	18.53	18.18	**17.88**	17.97	18.01	18.16	19.53
Ochiai	19.43	18.25	**17.97**	18.16	18.36	18.56	18.57	18.78	20.18
JacCube	21.18	19.76	18.88	**18.60**	18.87	18.99	18.98	19.16	20.51
Zoltar	19.46	19.37	19.24	19.56	19.87	19.71	19.41	**19.16**	20.52
Wong4	21.02	20.79	20.44	20.39	19.87	19.21	**19.17**	19.36	20.78
M2	21.72	20.24	19.22	19.06	**19.07**	19.22	19.18	19.36	20.80
AMean	19.95	18.71	**18.50**	18.77	19.15	19.34	19.33	19.54	20.97
Pearson	20.00	18.70	**18.47**	18.77	19.10	19.36	19.35	19.56	21.09
Jaccard	19.90	18.59	**18.56**	18.96	19.14	19.31	19.35	19.69	21.10
Ample2	20.00	18.60	**18.45**	18.66	19.36	19.58	19.58	19.87	21.37
Rogot2	20.00	18.68	**18.54**	18.71	18.84	18.95	18.94	20.23	21.78
CBI Log	19.60	**19.06**	19.21	19.76	20.17	20.25	20.39	21.47	22.26
Wong3	23.60	23.14	22.39	22.15	21.77	21.09	**21.02**	21.13	22.54
O^p	23.93	23.60	23.46	22.83	22.44	21.34	**21.30**	21.43	22.84
Tarantula	22.11	**21.93**	22.16	22.43	22.69	22.69	22.74	22.85	23.13
Ample	22.71	21.46	**21.40**	21.69	22.43	22.70	22.71	22.95	24.49
O	50.43	48.55	48.61	47.73	47.13	46.55	43.52	37.03	**24.95**
Russell	26.46	26.27	**26.13**	26.18	26.36	26.47	26.44	26.50	32.10
Binary	50.41	48.56	48.62	47.82	47.32	46.86	44.16	37.96	**34.02**
Overlap	**27.68**	28.04	**27.68**	28.04	27.88	29.02	28.58	33.90	34.02

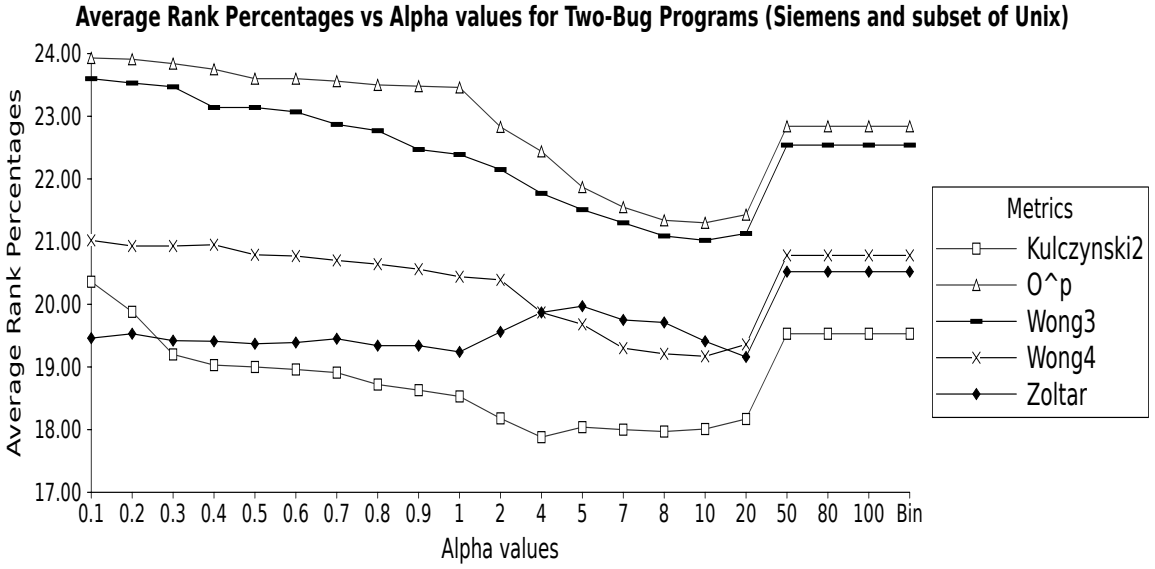

Figure 9.3: Average Rank Percentages for the Different α values of the Two-Bug Siemens Test Suite and the subset of the Unix Test Suite

Table 9.6: Average Rank Percentages for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
Kulczynski2	17.98	16.37	**16.14**	16.78	16.86	17.80	17.86	18.53	21.94
McCon	17.98	16.36	**16.14**	16.78	16.87	17.79	17.85	18.53	21.94
AMean	17.78	**16.50**	16.55	16.75	17.07	18.17	18.09	18.84	22.24
Pearson	17.73	**16.61**	16.63	16.86	17.12	18.20	18.13	18.87	22.31
Rogot2	17.86	16.75	**16.71**	16.93	17.11	18.10	18.01	19.09	22.53
Ample2	18.13	16.64	**15.92**	15.96	17.30	18.25	18.30	19.06	22.56
Ochiai	17.51	16.38	**16.30**	17.24	17.43	18.41	18.44	19.16	22.60
Wong4	16.96	**16.79**	16.98	17.35	18.11	18.34	18.34	19.16	22.60
Jaccard	18.09	**16.56**	16.57	17.48	17.80	18.74	18.81	19.61	23.12
CBI Log	15.43	**15.19**	15.55	17.23	18.14	18.72	19.08	21.31	23.37
Zoltar	**16.18**	16.35	17.06	18.30	18.65	19.70	19.58	20.06	23.49
Ample	18.92	17.48	**16.78**	16.90	18.32	19.30	19.34	20.08	23.88
JacCube	18.93	18.25	**17.99**	18.06	19.86	20.74	20.81	21.48	24.86
M2	19.10	18.29	**17.99**	18.91	19.86	20.89	20.88	21.57	24.97
Wong3	19.83	19.79	19.44	**19.31**	20.95	21.36	21.29	21.92	25.24
O^p	19.98	20.09	20.32	**19.82**	21.38	21.37	21.35	22.00	25.33

Continued on next page

Table 9.6 – continued from previous page

Metric	0.1	0.5	1	2	4	8	10	20	Bin
Tarantula	**25.35**	25.75	26.42	26.66	26.91	26.94	26.95	26.98	27.23
Russell	**20.05**	20.22	20.58	21.36	22.16	23.36	23.40	24.26	28.67
O	50.11	40.66	39.26	38.22	38.77	37.29	32.57	31.69	**29.29**
Binary	50.08	40.68	39.22	38.29	38.96	37.56	33.44	32.71	**32.43**
Overlap	**33.69**	35.24	35.69	36.35	36.01	37.68	37.47	38.08	38.34

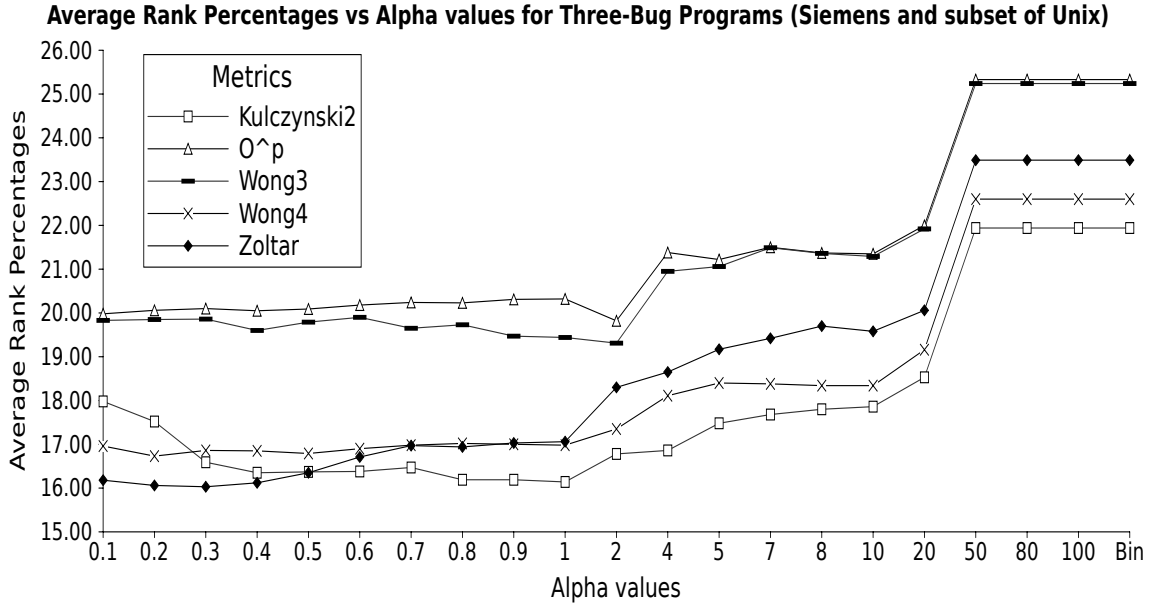

Figure 9.4: Average Rank Percentages for the Different α values of the Three-Bug Siemens Test Suite and the subset of the Unix Test Suite

Table 9.7: Average Rank Percentages (on average of 10 bins) for the Multiple-bug Space Programs with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
Zoltar	4.01	4.46	4.28	4.15	3.33	2.70	**2.34**	2.38	2.42
JacCube	2.76	2.79	2.74	**2.39**	2.46	2.46	2.46	2.45	2.49
O	50.10	50.55	50.72	50.66	47.25	42.26	41.63	41.06	**2.54**
O^p	6.18	6.50	5.90	4.94	3.82	**2.44**	2.51	2.51	2.54
Wong3	5.40	5.42	4.69	4.30	3.76	**2.45**	2.52	2.52	2.54

Continued on next page

Table 9.7 – continued from previous page

Metric	0.1	0.5	1	2	4	8	10	20	Bin
M2	3.38	3.21	2.76	**2.49**	**2.49**	2.54	2.54	2.53	2.58
Kulczynski2	3.49	3.29	3.17	2.96	**2.60**	2.61	2.61	**2.60**	2.64
McCon	3.49	3.29	3.17	2.96	**2.60**	2.61	2.61	**2.60**	2.64
Wong4	5.13	4.94	4.69	4.31	3.52	**2.61**	2.65	2.68	2.82
Ochiai	2.85	2.78	2.75	**2.71**	2.78	2.79	2.79	2.78	2.85
Jaccard	**3.01**	3.06	3.09	3.18	3.29	3.30	3.30	3.29	3.37
Rogot2	3.51	3.55	3.53	3.55	3.63	3.39	3.37	**3.36**	3.42
Pearson	3.45	3.45	3.43	3.43	3.51	3.43	**3.42**	**3.42**	3.49
Ample2	3.59	3.56	3.49	**3.40**	3.42	3.47	3.47	3.47	3.52
AMean	**3.49**	3.50	3.50	3.51	3.59	3.58	3.58	3.57	3.65
CBI Log	3.91	3.93	3.84	3.82	3.79	**3.76**	**3.76**	3.77	3.85
Tarantula	**4.38**	4.39	4.39	**4.38**	4.40	4.40	4.40	4.43	4.44
Ample	8.21	8.18	8.11	**8.00**	**8.00**	8.05	8.05	8.04	8.09
Russell	**10.25**	10.46	10.42	10.42	10.31	10.31	10.37	12.52	17.85
Binary	50.10	50.55	50.72	50.68	47.42	42.07	41.76	**17.85**	**17.85**
Overlap	6.54	6.48	**6.31**	6.38	6.54	16.02	15.87	15.70	18.07

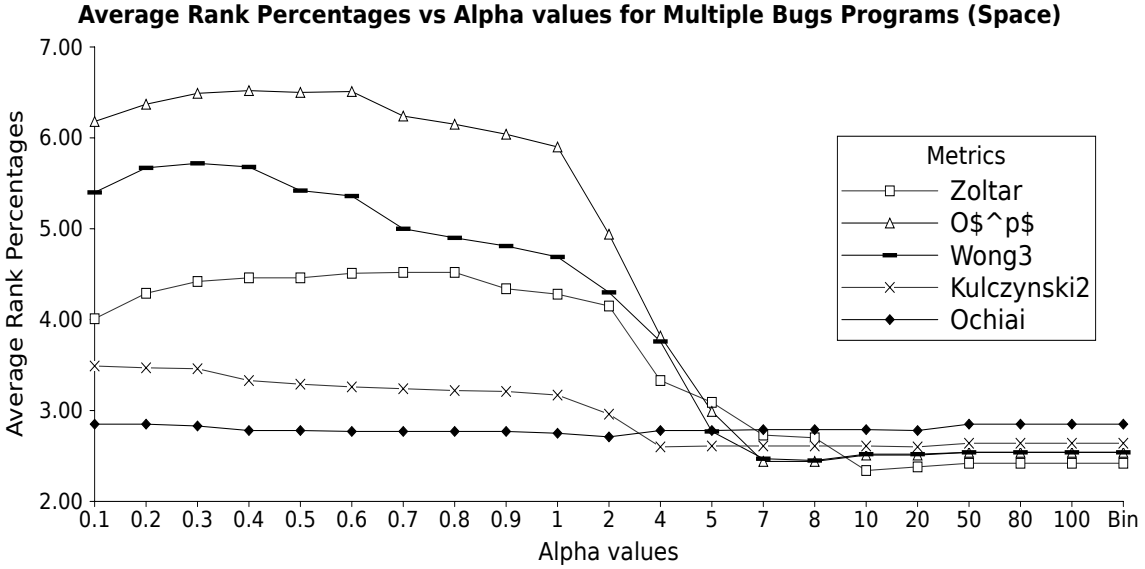

Figure 9.5: Average Rank Percentages for the Different α values of the Multiple-bug Space Programs

We evaluate the traditional binary weighting approach for the two-bug and three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite in Section 5.9

of Chapter 5. Kulczynski2 and McCon metrics are the most effective metrics to be used in these benchmarks. By using our proposed approach (using frequency weighting function) on these benchmarks, we observe that the performance of bug localization for the Kulczynski2 and McCon metrics improve as compared to using the traditional binary weighting approach (Bin) (see Table 9.5 and Table 9.6). By using Kulczynski2 and McCon metrics, we observe the most improvement of bug localization performance using our proposed approach with the α value of 4 and 1 as compared to using the traditional binary weighting approach in Table 9.5 and Table 9.6 respectively. In these tables, other metrics such as Ochiai and Ample show the best improvement in bug localization performance by using our proposed approach with the α value of 1 compared with using the traditional binary weighting approach (Bin).

We also report the *Subset* of multiple-bug Space programs using our proposed approach for the different α variants in Table 9.7. There is some improvement of bug localization performance using our proposed approach for metrics, such as Zoltar, O^p, Wong3, and M2, to name a few. For Zoltar, it performs the best when evaluating our approach with the α values of 10 compared to using the traditional binary weighting approach (Bin) with the average rank percentages of 2.34% and 2.42% respectively.

For the two-bug Siemens Test Suite and the subset of the Unix Test Suite, metrics in Figure 9.3 show that the bug localization performance improves using our proposed approach with the wider range of the α values of between 2 to 50 compared with the traditional binary weighting approach (Bin). We observe any α value less than 50 already shows improved bug localization performance for all the metrics in Figure 9.4, using our proposed approach on the three-bug Siemens Test Suite and the subset of the Unix Test Suite. For multiple-bug Space programs in Figure 9.5, we observe that within a range of the α values between 10 to 50, our proposed approach already shows improved bug localization performance as compared to using the traditional binary weighting approach (Bin).

In all these tables, we do not observe any improvement of bug localization performance on O metric using our proposed frequency weighting function approach. Any spectra metric with higher average rank percentages is not useful. In our evaluation, spectra metrics located at the bottom of Table 9.4 – 9.7 and Table G.1 – G.2 should not be considered at the first place by the programmer in locating bugs.

9.3.3 Statistical Significance

In Subsection 9.3.1 and Subsection 9.3.2, we observe that bug localization performance improves using our proposed frequency weighting function approach compared with the traditional binary weighting approach. It is unclear whether the improvement of bug

localization performance of our proposed approach is significant from statistical point of view.

In our evaluation, we observe the different ranges of the α variants that show the bug localization performance improves using our proposed approach compared to using the traditional binary weighting approach. The different ranges of the α variants that show improvement of bug localization performance using our proposed approach might be program dependent. In practice, the programmer does not know the number of bugs in the program. From all the observations in the tables, we can generalise that for any program without any knowledge on the number of bugs, choosing an α value of 10 is sufficient to show improvement in bug localization performance using our proposed approach compared to using the traditional binary weighting approach. Therefore, we establish a hypothesis, *by using our proposed frequency weighting function approach (with α value of 10), the bug localization performance improves as compared to using the traditional binary weighting approach.*

We perform the one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] on all the spectra metrics to check the statistical significance of the improved bug localization performance using our proposed approach (α value is 10), compared to the traditional binary weighting approach. For Space, we showed that the bug localization performance of the *Subset* of Space programs (subset of the Space test cases) across the 10 bins is very similar (see Figure 5.13 and Figure 5.16). Therefore, we only perform the statistical test on one of the 10 bins of the subset of the Space programs to test the significance of our hypothesis.

Significance of Improved Bug Localization Performance on Single Bug Programs

For the Siemens Test Suite and the subset of the Unix Test Suite, better performing metrics, namely O^p, Wong3, and Zoltar metrics in Table 9.4 have the p-value [Rice, 1989] for our hypothesis of 0.0069, 0.0112, and 0.0124 respectively. A p-value of less than 0.05 indicates that the improved bug localization performance using our proposed approach (α value of 10), is statistically significant as compared to the traditional binary weighting approach with confidence greater than 95%.

For Concordance and Space (Table G.1 and G.2), we observe marginal improvement of bug localization performance on several metrics. We do not observe statistically significant improvement in bug localization performance using our proposed approach (α value of 10) compared to the traditional binary weighting approach.

Significance of Improved Bug Localization Performance on Multiple-bug Programs

For the multiple-bug programs (Table 9.5, Table 9.6, and Table 9.7), we perform significant tests on better performing metrics using our hypothesis established earlier. Most of the better performing metrics in these tables have the p-value [Rice, 1989] of our hypothesis of < 0.01. Therefore, these metrics show a statistically significant improvement in bug localization performance using our proposed approach (α value of 10) with confidence greater than 99%.

9.4 Summary

In this chapter, we proposed the use of extra information, which is the frequency counts of test coverage to map into the respective spectra properties a_{np}, a_{nf}, a_{ep}, and a_{ef} using a frequency weighting function. Frequency counts of test coverage indicate how *many* times the respective statement of the test case has been executed. From our evaluation on benchmarks (Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space programs), exploiting frequency counts information led to significant improvements in bug localization performance compared to using the traditional binary weighting approach.

We showed that by using the proposed frequency weighting function within the range of α values of 10 to 20 on most of the better performing metrics generally improves the bug localization performance, compared to using the traditional binary weighting approach. We chose α of 10 and showed statistically significant improvement in bug localization performance for most of the better performing metrics on all the multiple-bug programs of our benchmarks. The improvement is statistically significant with confidence greater than 99% and is in the range of average rank percentages of 0.03% to 4.52%.

10 Conclusions

This chapter concludes the thesis. There are two sections in this chapter. The first section briefly summarises the contributions of the thesis in the spectral debugging area. The second section explores several avenues for future research.

10.1 Summary

In this thesis, we have made the following contributions to spectral debugging:

- A comprehensive study of spectra metrics in the literature and the application of these metrics in spectral debugging.
- The use of the model-based approach based on a simple if-then-else program ($ITE2_8$) to understand single bug programs, and the proposal of optimal spectra metrics O and O^p for improving the bug localization performance of single bug programs.
- The relationship of bug consistency, q_e, with respect to bug localization performance.
- Bug localization performance using unique (non-redundant) test cases.
- Varying weights on fail tests and the proposed incremental ranking approaches of the top-ranked statements.
- The use of more information of test coverage, such as the frequency counts of test execution.

The major results obtained in each part of the thesis are indicated as follows.

10.1.1 Model Program

We have advanced the state-of-the-art of software diagnosis using program spectra. Using a model-based approach (the simple model program $ITE2_8$), we are able to obtain a better understanding of single bug programs. This enables us to develop optimal spectra metrics, O and O^p, for single bug programs. We studied a comprehensive list of spectra metrics used in the literature. Our experiments on the model program $ITE2_8$ fit well with the overall evaluation on the real world benchmarks (Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space). We showed improvement in bug localization performance using the O and O^p metrics. These metrics outperformed other metrics proposed in the debugging area such as the Wong3, Wong4, Zoltar, Ochiai, Jaccard, Tarantula, and Ample metrics.

10.1.2 Bug Consistency

We studied the relationship between bug consistency, q_e, and bug localization performance. Bug consistency refers to how consistent the buggy statement shows an unintended output when executed by test cases. When the bug is not consistent (q_e is small), bug localization performance may vary. As the bug is more consistent (q_e approaching 1), bug localization performance improves. However, we observed ties in the program code affected bug localization performance even when the bug of the program is consistent (q_e is 1).

10.1.3 Equivalence of Spectra Metrics

We showed that any two spectra metrics are equivalent for ranking if and only if any one of them can be transformed to the other using a monotonically increasing function (see Lemma 5.2.1). For example, the Russell and Rao (Russell) and Wong1 metrics are equivalent for ranking. The Jaccard, Anderberg, Sneath & Sokal 2, Sørensen-Dice, Dice, Goodman, Levandowsky, and Kulczynski1 metrics are also equivalent for ranking. We also showed that the bug consistency, q_e, CBI Increase (CBI Inc), Coef, and Tarantula [Jones and Harrold, 2005] metrics are equivalent.

10.1.4 Unique (Non-redundant) Test Cases

We proposed using unique (non-redundant) test cases in order to evaluate the bug localization performance of programs. We found redundant test cases in the test suites of our

benchmarks (Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space). We performed an empirical study of bug localization performance using unique test cases on the single bug and multiple-bug programs. We generally observed no degradation of bug localization performance using unique test cases on better performing metrics for most of the benchmarks.

We also showed the importance of using a larger number of unique test cases to improve bug localization performance. We found improvement in bug localization performance for better performing metrics using all the unique test cases, compared to using 10% of the unique pass and fail test cases on all our benchmarks. The latter improvement for most of our benchmarks is statistically significant with confidence greater than 95%. We found the improvement for the better performing metrics on all the multiple-bug programs of our benchmarks is in the range of 5.83% to 11.45%.

10.1.5 Varying Weights on Fail Tests

We proposed to vary the weights for different fail test cases. Program statements executed in a fail test with fewer statements are more likely buggy than those program statements executed in a fail test that executed a larger number of statements. We assigned different weights to the statements according to the information of the fail tests. This is known as the Weighted approach. For multiple-bug Siemens Test Suite and the subset of the Unix Test Suite programs, we observed a statistically significant improvement in bug localization performance on all better performing metrics using the Weighted approach as compared to the Unweighted approach (without assigning any weights). The improvement is statistically significant with confidence greater than 95% and ranging in the average rank percentages of 0.06% to 0.98%.

Apart from assigning weights to the fail tests, we also proposed to rank the top statements of the program incrementally. For each iteration, once the top-ranked statement has been decided, the other statements can be ranked under the assumption that the top-ranked statement is not buggy. We proposed to rank the top 10% and 20% of the top-ranked statements of a program (and use the Weighted approach for the remainder). We observed improvement using these approaches as compared to the Unweighted approach especially on the multiple-bug programs.

We evaluate the top-down incremental ranking approach where we rank all the top ranked statements until there is a fail test that does not execute any statement. For single bug programs, we only found a marginal improvement in bug localization performance on better performing metric, O^p, by using the top-down incremental ranking approach as compared to using the Unweighted approach. For the multiple-bug programs, we found a statistically significant improvement in bug localization performance on all the better per-

forming metrics using the top-down incremental ranking approach compared to the Unweighted approach. The improvement is statistically significant with confidence greater than 95% and ranging from the average rank percentages of 0.41% to 4.63%.

We validated the proposed Weighted and top-down incremental ranking approaches on the unique test cases of our benchmarks in Appendix F. The improvement of bug localization performance using the proposed Weighted and incremental ranking approaches on the benchmarks further validated the robustness of our proposed approaches.

10.1.6 Frequency Weighting Function

Instead of using the binary test coverage (*traditional binary weighting approach*), we proposed the use of more information of the test coverage, namely the frequency counts. A frequency weighting function is applied by using a sigmoid function to map the frequency counts to the respective program spectra properties $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$. We showed improvement in bug localization performance on better performing metrics using the frequency weighting function compared to using the traditional binary weighting approach on our benchmarks (Siemens Test Suite, subset of the Unix Test Suite, Concordance, and Space).

By using the frequency weighting function with the range of α value of 10 to 20, we observed improvement in bug localization performance compared to using the traditional binary weighting approach on most of the better performing metrics for all of our benchmarks. Therefore, we chose the α value of 10 for the frequency weighting function. We observed improvement of bug localization performance compared to using the traditional binary weighting on most of the better performing metrics of all single bug programs. The improvement is in the range of average rank percentages from 0.02% to 0.86%. For all the multiple-bug programs, we observed a statistically significant improvement on most of the better performing metrics (ranging from 0.03% to 4.52%) with confidence greater than 99%.

10.2 Future Directions

Some future directions which could be investigated are outlined below:

- We can apply our proposed optimal spectra metrics, O and O^p, to the predicate-based CBI system [Liblit et al., 2005], and the Holmes system [Chilimbi et al., 2009], which uses path-based spectra coverage. The bug localization performance of these metrics can be compared with other metrics, such as $CBIInc$ and $CBILog$, which have been proposed in the CBI and Holmes systems. Recently, we proposed

to reconstruct predicate-based spectra coverage based on statement-based spectra coverage [Naish et al., 2010]. Several predicate-based spectra metrics have been proposed and bug localization performance for single bug programs improved using these metrics, as compared to using the O^p metric (the optimal metric for single bug programs). These predicate-based metrics could potentially be applied to the Holmes system [Chilimbi et al., 2009] to observe bug localization performance of the path-based spectra coverage.

- We can extend the study of the model-based approach to multiple-bug programs, although the assumptions we have made for single bug programs are not applicable to multiple-bug programs. For example, in a typical multiple-bug program, the buggy statements are not necessarily executed by all the fail test cases. Further studies are needed to adjust different parameters on the models of multiple-bug programs. An in-depth understanding of a particular multiple-bug model program would be useful in developing an optimal metric for multiple-bug programs.

- The history log provides information on what has been fixed in the past, and which particular statements of the program were affected in the past. Moin et al. used the information of history logs (bug information, changes made on the program code) of previous projects stored in revision control system to locate bugs [Moin and Khansari, 2010]. They apply a machine learning approach, which is support vector machine [Steinwart and Christmann, 2008] to train and predict the location of bug in the program code stored in the revision control system. We can apply this approach by incorporating this extra information together with the ranked program statements from the program spectra. Statements of the program not altered before are less likely to be buggy when compared to statements altered before by the programmers. Therefore, programmers can focus on top-ranked statements that have been recently fixed by the programmer to narrow down the search of the bugs.

- Typically in software testing, test cases are developed to ensure the test coverage of the program is achieved (program statements are covered and requirements of the program are tested). However, we believe, for debugging purposes, building distinct test cases is important, even though the test coverage might be small and we propose this as future work.

We hope that this thesis will become the basis for much further analysis of this area of spectral debugging.

Bibliography

[ABC, 2008] ABC. *Years of Failure Caused M5 Mess: RTA Veteran*. Retrieved from http://www.abc.net.au/news/stories/2008/09/23/2371848.htm.

[ABC, 2010] ABC. *Myki Customers Caught in Programming Error*. Retrieved from http://www.abc.net.au/news/stories/2010/03/26/2856820.htm?section=business.

[Abreu et al., 2006] Abreu, R., Zoeteweij, P., and van Gemund, A. (2006). An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46. IEEE Computer Society Washington, DC, USA.

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and van Gemund, A. (2007). On the Accuracy of Spectrum-based Fault Localization. *Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation, 2007. TAICPART-Mutation 2007*, pages 89–98.

[ACCC, 2010] ACCC. *Electronic Funds Transfer at Point of Sale (EFT-POS)*. Retrieved from http://www.accc.gov.au/content/index.phtml/itemId/816401.

[A.Gonzalez, 2007] A.Gonzalez (2007). Automatic Error Detection Techniques based on Dynamic Invariants. Master's thesis, Delft University of Technology, The Netherlands.

[Agrawal and Horgan, 1990] Agrawal, H. and Horgan, J. (1990). Dynamic Program Slicing. *ACM SIGPLAN Notices*, 25(6):246–256.

[Agrawal et al., 1995] Agrawal, H., Horgan, J., London, S., Wong, W., and Bellcore, M. (1995). Fault Localization using Execution Slices and Dataflow Tests. In *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 143–151.

[Ali et al., 2009] Ali, S., Andrews, J., Dhandapani, T., and Wang, W. (2009). Evaluating the Accuracy of Fault Localization Techniques. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 76–87. IEEE Computer Society.

[Anderberg, 1973] Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Monographs and Textbooks on Probability and Mathematical Statistics. Academic Press, Inc., New York.

[Andrews et al., 2005] Andrews, J., Briand, L., and Labiche, Y. (2005). Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411. ACM.

[BankWest, 2010] BankWest. *Bank of Western Australia (Bankwest)*. Retrieved from http://www.bankwest.com.au/.

[Beck and Eichmann, 2002] Beck, J. and Eichmann, D. (2002). Program and Interface Slicing for Reverse Engineering. In *Proceedings of the 15th International Conference on Software Engineering*, pages 509–518. IEEE Computer Society.

[Beizer, 1995] Beizer, B. (1995). *Black-box Testing*. Wiley New York.

[Bellard, 2010] Bellard, F. *Tiny C Compiler*. Retrieved from `http://bellard.org/tcc/`.

[Berkhin, 2006] Berkhin, P. (2006). Survey of Clustering Data Mining Techniques. *Grouping Multidimensional Data: Recent Advances in Clustering*, pages 25–71.

[Berlekamp, 1968] Berlekamp, E. (1968). *Algebraic Coding Theory*, volume 111. McGraw-Hill New York.

[Bloom, 1981] Bloom, S. (1981). Similarity Indices in Community Studies: Potential Pitfalls. *Mar. Ecol. Prog. Ser*, 5(2):125–128.

[BOQ, 2010] BOQ. *Bank of Queensland (BOQ)*. Retrieved from `http://www.boq.com.au/`.

[Boser et al., 1992] Boser, B., Guyon, I., and Vapnik, V. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory*, pages 144–152. ACM.

[Braun-Blanquet, 1932] Braun-Blanquet, J. (1932). Plant Sociology: The Study of Plant Communities. *Trans. by Fuller, GD & Conard*.

[Breiman, 2001] Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

[Briand et al., 2007] Briand, L., Labiche, Y., and Liu, X. (2007). Using Machine Learning to Support Debugging with Tarantula. In *Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, pages 137–146. IEEE Computer Society.

[Brin et al., 1997] Brin, S., Motwani, R., Ullman, J., and Tsur, S. (1997). Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 255–264. ACM.

[Bross, 1977] Bross, I. (1977). Critical Levels, Statistical Language and Scientific Inference. *Experimental Design and Interpretation*, 6:476.

[Charette, 2010] Charette, R. N. *Y2K Bug Ten Years (Plus Six) Late*. Retrieved from `http://spectrum.ieee.org/riskfactor/computing/software/y2k-bug-ten-years-plus-six-late`.

[Chen, 2009] Chen, B. X. *Snow Leopard Update Fixes Deletion Bug*. Retrieved from `http://www.wired.com/gadgetlab/2009/11/snowleopard-update/`.

[Chen et al., 2002] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC. IEEE Computer Society.

[Chen and Cheung, 1993] Chen, T. and Cheung, Y. (1993). Dynamic Program Dicing. In *Proceedings of the International Conference on Software Maintenance (CSM-93)*, pages 378–385.

[Chilimbi et al., 2009] Chilimbi, T., Liblit, B., Mehra, K., Nori, A., and Vaswani, K. (2009). HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society.

[Chung et al., 2008] Chung, Y., Huang, C., and Huang, Y. (2008). A Study of Modified Testing-based Fault Localization Method. In *Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 168–175. IEEE Computer Society.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering*, volume 27, pages 342–351, Missouri, USA. ACM Press New York, NY, USA.

[CNT, 2010] CNT. *CNT*. Retrieved from `http://sourceforge.net`.

[Cohen, 1960] Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46.

[Cohen, 1995] Cohen, W. (1995). Fast Effective Rule Induction. In *Proceedings of the 12th International Conference on Machine Learning, Tahoe City, California*, pages 115–123. Morgan Kaufmann.

[Dahm et al., 2002] Dahm, M., van Zyl, J., and Haase, E. *Bytecode Engineering Library (BCEL)*. Retrieved from `http://jakarta.apache.org/bcel/`.

[Dallmeier et al., 2005] Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight Bug Localization with AMPLE. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging*, pages 99–104. ACM.

[Debroy et al., 2010] Debroy, V., Wong, W., Xu, X., and Choi, B. (2010). A Grouping-based Strategy to Improve the Effectiveness of Fault Localization Techniques. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*, pages 13–22. IEEE Computer Society.

[Dennette A. Harrod, 1996] Dennette A. Harrod, J. *The First Computer Bug!* Retrieved from `http://www.waterholes.com/~dennette/1996/hopper/bug.htm`.

[Di Fatta et al., 2006] Di Fatta, G., Leue, S., and Stegantova, E. (2006). Discriminative Pattern Mining in Software Fault Detection. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, pages 62–69. ACM.

[Dickinson et al., 2001] Dickinson, W., Leon, D., and Podgurski, A. (2001). Finding Failures by Cluster Analysis of Execution Profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348. IEEE Computer Society Washington, DC, USA.

[Do et al., 2005] Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. In *Empirical Software Engineering*, pages 405–435. Springer.

[Dodd, 1938] Dodd, E. (1938). Definitions and Properties of the Median, Quartiles, and other Positional Means. *American Mathematical Monthly*, 45(5):302–306.

[Domingos, 1999] Domingos, P. (1999). Metacost: A General Method for Making Classifiers Cost-Sensitive. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 155–164. ACM.

[Duarte et al., 1999] Duarte, J., Santos, J., and Melo, L. (1999). Comparison of Similarity Coefficients based on RAPD Markers in the Common Bean. In *Genetics and Molecular Biology*, volume 22, pages 427–432. SciELO Brasil.

[Dunham, 2002] Dunham, M. (2002). *Data mining: Introductory and Advanced Topics*. Prentice Hall, PTR Upper Saddle River, NJ, USA.

[eibe, 2010] eibe, fracpete, w. *Weka—Machine Learning Software in Java*. Retrieved from `http://sourceforge.net/projects/weka/`.

[Emerson, 2008] Emerson, D. *Traffic Chaos Hits Sydney Again*. Retrieved from `http://www.smh.com.au/news/national/traffic-chaos-hits-sydney-again/2008/06/25/1214073288063.html?page=fullpage#contentSwap1`.

[Etcheberry, 1977] Etcheberry, J. (1977). The Set-Covering Problem: A New Implicit Enumeration Algorithm. *Operations Research*, 25(5):760–772.

[Everett and McLeod, 2007] Everett, G. and McLeod, R. (2007). *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society.

[Everitt, 1978] Everitt, B. (1978). *Graphical Techniques for Multivariate Data*. North-Holland, New York.

[Everitt and Rabe-Hesketh, 1997] Everitt, B. and Rabe-Hesketh, S. (1997). *The Analysis of Proximity Data*. Arnold Publishers.

[Fager and McGowan, 1963] Fager, E. and McGowan, J. (1963). Zooplankton Species Groups in the North Pacific. *Science*, 140:453–460.

[Ferrante et al., 1987] Ferrante, J., Ottenstein, K., and Warren, J. (1987). The Program Dependence Graph and Its Use in Optimisation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349.

[Festa, 2001] Festa, P. *Governments Push Open-Source Software*. Retrieved from `http://news.cnet.com/2100-1001-272299.html`.

[Fleiss, 1965] Fleiss, J. (1965). Estimating the Accuracy of Dichotomous Judgments. *Psychometrika*, 30(4):469–479.

[Forbes, 1933] Forbes, W. (1933). A Grouping of the Agrotine Genera. *Entomologica Americana*, 14(1).

[Frank and Witten, 1998] Frank, E. and Witten, I. (1998). Generating Accurate Rule Sets Without Global Optimisation. pages 144–151.

[Frankl and Iakounenko, 1998] Frankl, P. and Iakounenko, O. (1998). Further Empirical Studies of Test Effectiveness. *ACM SIGSOFT Software Engineering Notes*, 23(6):153–162.

[Friedman et al., 2001] Friedman, J., Tibshirani, R., and Hastie, T. (2001). *The Elements of Statistical Learning*. Springer-Verlag New York.

[Gartner, 2010] Gartner. *Gartner Report IT Spending 2010*. Retrieved from `http://www.slideshare.net/rsink/gartner-report-it-spending-2010`.

[GCC, 2010a] GCC, G. *GCC, The GNU Compiler Collection*. Retrieved from `http://gcc.gnu.org/`.

[GCC, 2010b] GCC, G. *GDB: The GNU Project Debugger*. Retrieved from `http://www.gnu.org/software/gdb/`.

[gimp, 2010] gimp. *GIMP - The GNU Image Manipulation Program*. Retrieved from `http://www.gimp.org/`.

[GNOME, 2010] GNOME. *GNOME Office/Gnumeric*. Retrieved from `http://projects.gnome.org/gnumeric/`.

[Goethals, 2003] Goethals, B. (2003). Survey on Frequent Pattern Mining. *Manuscript*, pages 1–43.

[Goodman and Kruskal, 1954] Goodman, L. and Kruskal, W. (1954). Measures of Association for Cross Classifications. *Journal of the American Statistical Association*, 49(268):732–764.

[Gower, 1971] Gower, J. (1971). A General Coefficient of Similarity and Some of its Properties. *Biometrics*, pages 857–871.

[Gower and Legendre, 1986] Gower, J. and Legendre, P. (1986). Metric and Euclidean Properties of Dissimilarity Coefficients. *Journal of Classification*, 3(1):5–48.

[Greenwood and Nikulin, 1996] Greenwood, P. and Nikulin, M. (1996). *A Guide to Chi-squared Testing*. Wiley-Interscience.

[Guyon et al., 2002] Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene Selection for Cancer Classification using Support Vector Machines. *Machine learning*, 46(1):389–422.

[Hailpern and Santhanam, 2002] Hailpern, B. and Santhanam, P. (2002). Software Debugging, Testing, and Verification. *IBM Systems Journal*, 41(1):4–12.

[Hall et al., 2009] Hall, M., Padua, D., and Pingali, K. (2009). Compiler Research: The Next 50 Years. *Communications of the ACM*, 52(2):60–67.

[Hamann, 1961] Hamann, U. (1961). Merkmalsbestand und Verwandtschaftsbeziehungen der Farinosae: Ein Beitrag zum System der Monokotyledonen. *Willdenowia*, 2(5):639–768.

[Hamming, 1950] Hamming, R. (1950). Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160.

[Hao et al., 2006] Hao, D., Zhang, L., Mei, H., and Sun, J. (2006). Towards Interactive Fault Localization using Test Information. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC)*, pages 277–284. IEEE Computer Society.

[Hao et al., 2008] Hao, D., Zhang, L., Pan, Y., Mei, H., and Sun, J. (2008). On Similarity-Awareness in Testing-based Fault Localization. *Automated Software Engineering*, 15(2):207–249.

[Hao et al., 2005] Hao, D., Zhang, L., Zhong, H., Mei, H., and Sun, J. (2005). Eliminating Harmful Redundancy for Testing-based Fault Localization using Test Suite Reduction: An Experimental Study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005*, pages 683–686. IEEE Computer Society.

[Harrold et al., 1993] Harrold, M., Gupta, R., and Soffa, M. (1993). A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):285.

[Heaton, 2008] Heaton, J. *Introduction to Neural Networks for Java*.

[Heimdahl and George, 2004] Heimdahl, M. and George, D. (2004). Test-suite Reduction for Model based Tests: Effects on Test Quality and Implications for Testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 176–185. IEEE Computer Society.

[Hollander and Wolfe, 1973] Hollander, M. and Wolfe, D. (1973). Nonparametric Statistical Methods. *New York*, page 518.

[Holliday et al., 2003] Holliday, J., Salim, N., Whittle, M., and Willett, P. (2003). Analysis and Display of the Size Dependence of Chemical Similarity Coefficients. *J. Chem. Inf. Comput. Sci*, 43(3):819–828.

[Horgan and London, 1992] Horgan, J. and London, S. (1992). ATAC: A Data Flow Coverage Testing Tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10.

[Horowitz et al., 1995] Horowitz, E., Sahni, S., and Mehta, D. (1995). *Fundamentals of Data Structures in C++*. Computer Science Press.

[Horwitz and Reps, 1992] Horwitz, S. and Reps, T. (1992). The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411. ACM.

[Hsu et al., 2008] Hsu, H., Jones, J., and Orso, A. (2008). RAPID: Identifying Bug Signatures to Support Debugging Activities. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 439–442. IEEE Computer Society.

[IEEE, 2004a] IEEE. *IEEE SA 1012-1998 - IEEE Standard for Software Verification and Validation*. Retrieved from `http://standards.ieee.org/reading/ieee/std_public/description/se/1012-1998_desc.html`.

[IEEE, 2004b] IEEE. *IEEE SA 829-1983 - IEEE Standard for Software Test Documentation*. Retrieved from `http://standards.ieee.org/reading/ieee/std_public/description/se/829-1983_desc.html`.

[IEEE, 2004c] IEEE. *IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990*. Retrieved from `http://www.amazon.com/Standard-Glossary-Engineering-Terminology-610-12-1990/dp/155937067X`.

[Jaccard, 1901] Jaccard, P. (1901). Étude Comparative de la Distribution Florale Dans une Portion des Alpes et des Jura. *Bull. Soc. Vaudoise Sci. Nat*, 37:547–579.

[Jackson, 2004] Jackson, E. *Software Problem Caused Cancer Patients' Radiation ODs*. Retrieved from `http://www.thepanamanews.com/pn/v_10/issue_01/science_01.html`.

[Jiang and Su, 2005] Jiang, L. and Su, Z. (2005). Automatic Isolation of Cause-effect Chains with Machine Learning. Technical report, CSE-2005-32, University of California, Davis.

[Jones et al., 2007] Jones, J., Bowring, J., and Harrold, M. (2007). Debugging in Parallel. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 16–26. ACM.

[Jones and Harrold, 2003] Jones, J. and Harrold, M. (2003). Test-suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*, pages 195–209.

[Jones and Harrold, 2005] Jones, J. and Harrold, M. (2005). Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. ACM.

[Jones et al., 2001] Jones, J., Harrold, M., and Stasko, J. (2001). Visualization for Fault Localization. In *Proceedings of the International Conference of Software Engineering Workshop on Software Visualization, Toronto, Ontario, Canada*, pages 71–75.

[Jones et al., 2002] Jones, J., Harrold, M., and Stasko, J. (2002). Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press New York.

[Kaesler, 1966] Kaesler, R. (1966). Quantitative Re-evaluation of Ecology and Distribution of Recent Foraminifera and Ostracoda of Todos Santos Bay, Baja California, Mexico.

[Klosgen, 1992] Klosgen, W. (1992). Problems for Knowledge Discovery in Databases and their Treatment in the Statistics Interpreter EXPLORA. *International Journal of Intelligent Systems*, 7(7):649–673.

[Kohonen, 2002] Kohonen, T. (2002). The Self-Organizing Map. In *Proceedings of the IEEE*, pages 1464–1480. IEEE Computer Society.

[Korel, 1988] Korel, B. (1988). PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260.

[Korel and Laski, 1990] Korel, B. and Laski, J. (1990). Dynamic Slicing of Computer Programs. *Journal of Systems and Software*, 13(3):187–195.

[Korel and Rilling, 1997] Korel, B. and Rilling, J. (1997). Dynamic Program Slicing in Understanding of Program Execution. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC97)*, pages 80–85. IEEE Computer Society.

[Krause, 1973] Krause, E. (1973). Taxicab Geometry. *Mathematics Teacher*, 66(8):695–706.

[Lance and Williams, 1966] Lance, G. and Williams, W. (1966). Computer Programs for Hierarchical Polythetic Classification (similarity analyses). *The Computer Journal*, 9(1):60–64.

[Lauritzen, 1996] Lauritzen, S. (1996). *Graphical Models*. Oxford University Press, USA.

[Lee, 1958] Lee, C. (1958). Some Properties of Nonbinary Error-correcting Codes. *IEEE Transactions on Information Theory*, 4(2):77–82.

[Lee et al., 2009a] Lee, H., Naish, L., and Ramamohanarao, K. (2009a). Study of the Relationship of Bug Consistency with respect to Performance of Spectra Metrics. In *Proceedings of the 2nd International Conference on Computer Science and Information Technology*, pages 501–508, Beijing, China. IEEE Computer Society.

[Lee et al., 2009b] Lee, H., Naish, L., and Ramamohanarao, K. (2009b). The Effectiveness of using Non Redundant Test Cases with Program Spectra for Bug Localization. In *Proceedings of the 2nd International Conference on Computer Science and Information Technology*, pages 127–134, Beijing, China. IEEE Computer Society.

[Lee et al., 2010] Lee, H., Naish, L., and Ramamohanarao, K. (2010). Effective Software Bug Localization using Spectral Frequency Weighting Function. In *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference (COMPSAC)*, pages 218–227. IEEE Computer Society.

[Levandowsky, 1972] Levandowsky, M. (1972). An Ordination of Phytoplankton Populations in Ponds of Varying Salinity and Temperature. *Ecology*, 53(3):398–407.

[Levandowsky and Winter, 1971] Levandowsky, M. and Winter, D. (1971). Distance between Sets. *Nature*.

[Liblit, 2004] Liblit, B. (2004). *Cooperative Bug Isolation*. PhD thesis, University of California.

[Liblit et al., 2003] Liblit, B., Aiken, A., Zheng, A., and Jordan, M. (2003). Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM International Conference on Programming Language Design and Implementation*, pages 141–154. ACM New York, NY, USA.

[Liblit et al., 2005] Liblit, B., Naik, M., Zheng, A., Aiken, A., and Jordan, M. (2005). Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26. ACM New York.

[Liu et al., 2005] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). SOBER: Statistical Model-based Bug Localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295.

[Lourenco et al., 2004] Lourenco, F., Lobo, V., and Bação, F. (2004). Binary-based Similarity Measures for Categorical Data and Their Application in Self-Organizing Maps. *JOCLAD*.

[Lucia et al., 2010] Lucia, Lo, D., Jiang, L., and Budi, A. (2010). Comprehensive Evaluation of Association Measures for Fault Localization. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–10. IEEE Computer Society.

[Lyle, 1985] Lyle, J. (1985). Evaluating Variations on Program Slicing for Debugging. *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, 46(5).

[Lyle and Weiser, 1987] Lyle, J. and Weiser, M. (1987). Automatic Program Bug Location by Program Slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877–882.

[Maxwell and Pilliner, 1968] Maxwell, A. and Pilliner, A. (1968). Deriving Coefficients of Reliability and Agreement for Ratings. *Br J Math Stat Psychol*, 21(1):105–16.

[McConnaughey and Laut, 1964] McConnaughey, B. and Laut, L. (1964). *The Determination and Analysis of Plankton Communities*. Lembaga Penelitian Laut.

[Meyer et al., 2004] Meyer, A., Garcia, A., Souza, A., and Souza Jr, C. (2004). Comparison of Similarity Coefficients used for Cluster Analysis with Dominant Markers in Maize (Zea mays L). *Genetics and Molecular Biology*, 27:83–91.

[Moin and Khansari, 2010] Moin, A. and Khansari, M. (2010). Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization. *Open Source Software: New Horizons*, pages 188–199.

[Morris and Cherry, 1983] Morris, R. and Cherry, L. (1983). DC- An Interactive Desk Calculator. *UNIX Time-sharing System: UNIX Programmer's Manual*, page 460.

[Mountford, 1962] Mountford, M. (1962). An Index of Similarity and its Application to Classificatory Problems. *Progress in Soil Zoology*, 43:50.

[Murtagh, 1983] Murtagh, F. (1983). A Survey of Recent Advances in Hierarchical Clustering Algorithms. *The Computer Journal*, 26(4):354.

[myki, 2010] myki. *Myki-It's Your Key*. Retrieved from `http://www.myki.com.au/`.

[Naish et al., 2009] Naish, L., Lee, H., and Ramamohanarao, K. (2009). Spectral Debugging with Weights and Incremental Ranking. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC)*, pages 168–175. IEEE Computer Society.

[Naish et al., 2010] Naish, L., Lee, H., and Ramamohanarao, K. (2010). Statements versus Predicates in Spectral Bug Localization. In *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC)*, pages 375–384. IEEE Computer Society.

[Naish et al., 2011] Naish, L., Lee, H., and Ramamohanarao, K. (2011). A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3).

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN*, pages 89–100. ACM Press New York, NY, USA.

[Neumann, 1990] Neumann, P. G. *Telephone World-The Crash of the AT&T Network in 1990*. Retrieved from `http://www.phworld.org/history/attcrash.htm`.

[Newman, 2002] Newman, M. *Software Errors Cost U.S. Economy $59.5 Billion Annually NIST Assesses Technical Needs of Industry to Improve Software-Testing*. Retrieved from `http://www.nist.gov/public_affairs/releases/n02-10.htm`.

[Newmark, 1988] Newmark, J. (1988). *Statistics and Probability in Modern Life*. Saunders College Pub.

[Neyman, 1937] Neyman, J. (1937). X-Outline of a Theory of Statistical Estimation based on the Classical Theory of Probability. *Phil. Trans. Royal Soc. London, A*, 236:333–380.

[Ng et al., 2001] Ng, A., Jordan, M., and Weiss, Y. (2001). On Spectral Clustering: Analysis and an Algorithm. *Advances in Neural Information Processing Systems 14*, pages 849–856.

[Nichols, 2010] Nichols, S. *Software Testing Market Set to Boom*. Retrieved from `http://www.v3.co.uk/v3/news/2268744/software-testing-set-boom#ixzz0xiyXK8jI`.

[Ochiai, 1957] Ochiai, A. (1957). Zoogeographic Studies on the Soleoid Fishes found in Japan and its Neighbouring Regions. *Bull. Jpn. Soc. Sci. Fish*, 22:526–530.

[Offutt et al., 1996] Offutt, A., Pan, J., Tewary, K., and Zhang, T. (1996). An Experimental Evaluation of Data Flow and Mutation Testing. *Software-Practice and Experience*, 26(2):165–176.

[Ostrand and Balcer, 1988] Ostrand, T. and Balcer, M. (1988). The Category-partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686.

[Oyster, 2010] Oyster. *What is Oyster?* Retrieved from `http://www.tfl.gov.uk/tickets/14836.aspx`.

[Pan and Spafford, 1992] Pan, H. and Spafford, E. (1992). Heuristics for Automatic Localization of Software Faults. Technical report, SERC-TR-116-P.

[Peters, 1968] Peters, J. (1968). A Computer Program for Calculating Degree of Biogeographical Resemblance between Areas. *Systematic Zoology*, 17(1):64–69.

[Pytlik et al., 2003] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. (2003). Automated Fault Localization using Potential Invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pages 273–276. ACM.

[Quinlan, 1993] Quinlan, J. (1993). *C4. 5: Programs for Machine Learning*. Morgan Kaufmann.

[Renieres and Reiss, 2003] Renieres, M. and Reiss, S. (2003). Fault Localization with Nearest Neighbor Queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39, Montreal, Canada. IEEE Computer Society.

[Reps et al., 1997] Reps, T., Ball, T., Das, M., and Larus, J. (1997). The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *ACM SIGSOFT Software Engineering Notes*, volume 22, pages 432–449, New York, USA. Springer-Verlag New York, Inc. New York.

[Rhythm, nd] Rhythm. *Rhythm*. Retrieved from `http://rhythm.sourceforge.net`.

[Rice, 1989] Rice, W. (1989). Analyzing Tables of Statistical Tests. *Evolution*, 43(1):223–225.

[Rogers and Tanimoto, 1960] Rogers, D. and Tanimoto, T. (1960). A Computer Program for Classifying Plants. *Science*, 132(3434):1115–1118.

[Rogot and Goldberg, 1966] Rogot, E. and Goldberg, I. (1966). A Proposed Index for Measuring Agreement in Test-Retest Studies. *Journal of Chronic Diseases*, 19(9):991–1006.

[Romanovskiĭ, 1970] Romanovskiĭ, V. (1970). *Discrete Markov Chains*. Wolters-Noordhoff.

[Rothermel et al., 1998] Rothermel, G., Harrold, M., Ostrin, J., and Hong, C. (1998). An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43. IEEE Computer Society.

[Russel and Rao, 1940] Russel, P. and Rao, T. (1940). On Habitat and Association of Species of Anopheline Larvae in South-Eastern Madras. *Journal of the Malaria Institute of India*, 3:153–178.

[Sahar and Mansour, 1999] Sahar, S. and Mansour, Y. (1999). An Empirical Evaluation of Objective Interestingness Criteria. In *SPIE Conference on Data Mining and Knowledge Discovery*, pages 63–74.

[Santelices and Harrold, 2007] Santelices, R. and Harrold, M. (2007). Efficiently Monitoring Data-flow Test Coverage. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 343–352. ACM.

[Santelices et al., 2009] Santelices, R., Jones, J., Yu, Y., and Harrold, M. (2009). Lightweight Fault-Localization using Multiple Coverage Types. In *Proceedings of the 31st International Conference on Software Engineering*, pages 56–66. IEEE Computer Society.

[Savage, 1960] Savage, J. (1960). Evolution of a Peninsular Herpetofauna. *Systematic Biology*, 9(3-4):184.

[Savageau, 1969] Savageau, M. (1969). Biochemical Systems Analysis+*:: II. The Steady-state Solutions for an N-Pool System using a Power-law Approximation. *Journal of Theoretical Biology*, 25(3):370–379.

[Savive, 2010] Savive. *Thera 25 Mistreatment*. Retrieved from `http://www.savive.com/casestudy/therac.html`.

[Scott, 1955] Scott, W. (1955). Reliability of Content Analysis: The Case of Nominal Scale Coding. *Public Opinion Quarterly*, 19(3):321–325.

[selinger, 2010] selinger. *CCRYPT- Download CCRYPT Software for Free at SourceForge.net*. Retrieved from `http://sourceforge.net/projects/ccrypt/`.

[Shapiro, 1987] Shapiro, F. (1987). Etymology of the Computer Bug: History and Folklore. *American Speech*, 62(4):376–378.

[Shiraishi and Savageau, 1992] Shiraishi, F. and Savageau, M. (1992). The Tricarboxylic Acid Cycle in Dictyostelium Discoideum. III. Analysis of Steady State and Dynamic Behavior. *Journal of Biological Chemistry*, 267(32):22926.

[Simpson, 1961] Simpson, G. (1961). Principles of Animal Taxonomy.[Animal Taxonomy]. *Columbia Biological Series*, 20.

[SIR, 2010] SIR. *Software-Artifact Infrastructure Repository*. Retrieved from `http://sir.unl.edu/php/index.php`.

[Smyth and Goodman, 1991] Smyth, P. and Goodman, R. (1991). Rule Induction using Information Theory. *Knowledge Discovery in Databases*, 1991.

[Sokal and Michener, 1975] Sokal, R. and Michener, C. (1975). A Statistical Method for Evaluating Systematic Relationships. *Multivariate Statistical Methods, Among-Groups Covariation*, pages 1409–1438.

[Sokal and Sneath, 1963] Sokal, R. and Sneath, P. (1963). *Principles of Numerical Taxonomy*. WH Freeman.

[Sørensen, 1948] Sørensen, T. (1948). A Method of Establishing Groups of Equal Amplitude in Plant Sociology based on Similarity of Species Content and its Application to Analyses of the Vegetation on Danish Commons. *K. danske vidensk. Selsk.*

[Sorgenfrei, 1958] Sorgenfrei, T. (1958). Molluscan Assemblages from the Marine Middle Miocene of South Jutland and their Environments, 1-2. *Danmarks Geologiske Undersogelse (2)*, 79:1–503.

[Steinwart and Christmann, 2008] Steinwart, I. and Christmann, A. (2008). *Support Vector Machines*. Springer Verlag.

[Stephenson et al., 1968] Stephenson, W., Williams, W., Lance, G., Institution, S., and Museum, U. S. N. (1968). *Numerical Approaches to the Relationships of Certain American Swimming Crabs (Crustacea: Portunidae)*. Smithsonian Inst.

[Tallam and Gupta, 2005] Tallam, S. and Gupta, N. (2005). A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42. ACM.

[Tan et al., 2002] Tan, P., Kumar, V., and Srivastava, J. (2002). Selecting the Right Interestingness Measure for Association Patterns. In *Proceedings of ACM SIGKDD ICKDM*, pages 32–41. ACM New York, NY, USA.

[Tan et al., 2004] Tan, P., Kumar, V., and Srivastava, J. (2004). Selecting the Right Objective Measure for Association Analysis. *Information Systems*, 29(4):293–313.

[Teitelbaum et al., 2001] Teitelbaum, T. et al. *Code Surfer User Guide and Reference, Technical Report, Gramma Tech Product Documentation, 2001.* Retrieved from `http://www.grammatech.com/csurf-doc/manual.html`.

[Telcordia Technologies, Inc., 1998] *Telecordia Software Visualization and Analysis Toolsuite (χSuds). Users Manual, Chapter 12*.

[Thomsett and Co, 1998] Thomsett, R. and Co, T. (1998). The Year 2000 bug: A Forgotten Lesson. *IEEE Software*, 15(4):91–93.

[Udny Yule and Kendall, 1948] Udny Yule, G. and Kendall, M. (1948). An Introduction to the Theory of Statistics. *C. Griffin and Co. Ltd., London*.

[Urbani, 1976] Urbani, C. (1976). A Numerical Analysis of the Distribution of British Formicidae1 (Hymenoptera, Aculeata). *Verhandlungen der Naturforschenden Gesellschaft in Basel*, page 51.

[Van Ommering et al., 2000] Van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2000). The Koala Component Model. *IEEE computer*, 78:85.

[van Rijsbergen, 1979] van Rijsbergen, C. J. (1979). *Information Retrieval*. Butterworth.

[Vapnik, 1998] Vapnik, V. (1998). *Statistical Learning Theory*, volume 2. Wiley New York.

[Walczak and Massart, 1996] Walczak, B. and Massart, D. (1996). The Radial Basis Functions–Partial Least Squares Approach as a Flexible Non-Linear Regression Technique. *Analytica Chimica Acta*, 331(3):177–185.

[Wang and Han, 2004] Wang, J. and Han, J. (2004). BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the 20th International Conference on Data Engineering*, page 79. IEEE Computer Society.

[Weiser and Lyle, 1986] Weiser, M. and Lyle, J. (1986). Experiments on Slicing-based Debugging Aids. In *Empirical Studies of Programmers*, pages 187–197.

[Weisstein, 2011] Weisstein, E. *Least Squares Fitting–Power Law*. Retrieved from `http://mathworld.wolfram.com/LeastSquaresFittingPowerLaw.html`.

[Williams, 2007] Williams. *Systems Glitch Hits Hundreds of Tokyo Stations*. washingtonpost.com, 2007. Retrieved from `http://www.washingtonpost.com/wp-dyn/content/article/2007/10/12/AR2007101200865_pf.html`.

[Wong et al., 2010] Wong, W., Debroy, V., and Choi, B. (2010). A Family of Code Coverage-based Heuristics for Effective Fault Localization. *Journal of Systems and Software*, 83(2):188–208.

[Wong et al., 1998] Wong, W., Horgan, J., London, S., and Mathur, A. (1998). Effect of Test Set Minimization on Fault Detection Effectiveness. *Software-Practice and Experience*, 28(4):347–369.

[Wong and Qi, 2004] Wong, W. and Qi, Y. (2004). An Execution Slice and Inter-block Data Dependency-based Approach for Fault Localization. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 366–373. IEEE Computer Society.

[Wong et al., 2007] Wong, W., Qi, Y., Zhao, L., and Cai, K. (2007). Effective Fault Localization using Code Coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 449–456, Washington DC, USA. IEEE Computer Society.

[Wong et al., 2008] Wong, W., Shi, Y., Qi, Y., and Golden, R. (2008). Using an RBF Neural Network to Locate Program Bugs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 27–36. IEEE Computer Society.

[Wong et al., 1994] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P. (1994). Effect of Test Set Minimization on the Fault Detection Effectiveness of the All-uses Criterion. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50.

[Xie et al., 2010] Xie, X., Chen, T. Y., and Xu, B. (2010). Isolating Suspiciousness from Spectrum-based Fault Localization Techniques. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*, pages 385–392. IEEE Computer Society.

[Yu et al., 2008] Yu, Y., Jones, J., and Harrold, M. (2008). An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proceedings of the 30th International Conference on Software Engineering*, pages 201–210. ACM.

[Yule, 1900] Yule, G. (1900). On the Association of Attributes in Statistics: With Illustrations from the Material of the Childhood Society. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, pages 257–319.

[Zeller, 2000] Zeller, A. (2000). From Automated Testing to Automated Debugging. *Uni Passau, Feb*.

[Zeller, 2002] Zeller, A. (2002). Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10. ACM Press New York, NY, USA.

[Zeller, 2010] Zeller, A. *AskIgor - Automated Debugging Service*. Retrieved from `http://www.st.cs.uni-saarland.de/askigor/`.

[Zhang et al., 2008] Zhang, Z., Jiang, B., Chan, W., and Tse, T. (2008). Debugging through Evaluation Sequences: A Controlled Experimental Study. In *Proceedings of the 32nd International Computer Software and Applications (COMPSAC'08)*, pages 128–135. IEEE Computer Society.

[Zheng et al., 2003] Zheng, A., Jordan, M., Liblit, B., and Aiken, A. (2003). Statistical Debugging of Sampled Programs. In *Advances in Neural Information Processing Systems 16*. Neural Information Processing Systems Foundation.

[Zheng et al., 2006] Zheng, A., Jordan, M., Liblit, B., Naik, M., and Aiken, A. (2006). Statistical Debugging: Simultaneous Identification of Multiple Bugs. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 1105–1112. ACM New York, NY, USA.

[Zimmermann and Zeller, 2002] Zimmermann, T. and Zeller, A. (2002). Visualizing Memory Graphs. *Software Visualization*, pages 533–537.

[Zoeteweij et al., 2007] Zoeteweij, P., Abreu, R., Golsteijn, R., and van Gemund, A. (2007). Diagnosis of Embedded Software using Program Spectra. In *Proceedings of the 14th International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 213–220. IEEE Computer Society.

Appendices

A

Reported Software Bug & Software Failure Incidents

Table A.1: Software Bug & Software Failure Incidents (partly taken from Charette [2005])

YEAR	COMPANY	OUTCOME / COST IN US$
2011	Apple	Alarm not working on the 1st Jan 2011. Caused customers to miss flights and to work. Suspected bug in the week-number algorithm.
2010	NAB	NAB system crashed. Delayed payments to customers. Due to the bug in the batch processing software code that contained instructions on how the system operate in the batch processing operations.
2010	Bank of Queensland [BOQ, 2010] and Bankwest in Australia [BankWest, 2010]	On 1st January, 2010, electronic point-of-sale transactions system (better known as EFTPOS) has switched the date automatically to year 2016. Retail businesses could not proceed with any EFTPOS transaction [ACCC, 2010] but to manually issue receipts to the customers [Charette, 2010].
2009	Apple	A bug found that replaced primary folder in Snow Leopard with empty folder when logged in as a *Guest* account. Caused loss of data stored in the primary folder such as photos, documents, and other type of files [Chen, 2009].
2005	Hudson Bay Co., Canada	Problems with inventory system contributed to $33.3 million[1] loss.
2004-05	UK Inland Revenue	Software errors contributed to $3.45 billion[1] tax-credit overpayment.
2004	Avis Europe PLC, UK	Enterprise Resource Planning (ERP) system cancelled after $54.5 million[2] is spent.

Continued on next page

Table A.1 – continued from previous page

YEAR	COMPANY	OUTCOME / COST IN US$
2004	Ford Motor Co.	Purchasing system abandoned after deployment cost approximately $400 million.
2004	J Sainsbury PLC, UK	Software did not have any record of the warehouse's existence. Supply-chain management system abandoned after deployment cost $527 million[2].
2004	Hewlett-Packard Co.	Problems with ERP system contributed to $160 million loss.
2003-04	AT & T Wireless	Customer relations management (CRM) upgrade problems led to revenue loss of $100 million.
2002	McDonald's Corp.	The Innovate information-purchasing system cancelled after $170 million is spent.
2002	Sydney Water Corp.	Billing system cancelled after $33.2 million[2] is spent.
2002	CIGNA Corp.	Problems with CRM system contributed to $445 million loss.
2001	Nike Inc.	Problems with supply-chain management system contributed to $100 million loss.
2001	KMart Corp.	Supply-chain management system is cancelled after $130 million is spent.
2000	Washington DC.	City payroll system abandoned after deployment cost $25 million.
1999	United Way	Administrative processing system cancelled after $12 million is spent.
1999	State of Mississippi	Tax system cancelled after $11.2 million is spent; state received $185 million damages.
1999	Hershey Foods Corp.	Problems with ERP system contributed to $151 million loss.
1998	Snap-on Inc.	Problems with order-entry system contributed to revenue loss of $50 million.
1997	U.S. Internal Revenue Service	Tax modernisation effort cancelled after $4 billion is spent.
1997	State of Washington	Department of Motor Vehicle (DMV) system cancelled after $40 million is spent.
1997	Oxford Health Plans Inc.	Billing and claims system problems contributed to quarterly loss; stock plummets, leased to $3.4 billion loss in corporate value.
1996	Arianespace, France	Software specification and design errors caused $350 million Ariane 5 rocker to explode.
1996	FoxMeyer Drug Co.	$40 million ERP system abandoned after deployment forced company into bankruptcy.

Continued on next page

Table A.1 – continued from previous page

YEAR	COMPANY	OUTCOME / COST IN US$
1995	Toronto Stock Exchange	Electronic trading system cancelled after $25.5 million[1] is spent.
1994	U.S. Federal Aviation Administration	Advanced Automation System cancelled after $2.6 billion is spent.
1994	State of California	DMV system cancelled after $44 million is spent.
1994	Chemical Bank	Software error caused a total of $15 million to be deducted from 100 000 customer accounts.
1993	London Stock Exchange	Taurus stock settlement system cancelled after $600 million[3] is spent.
1993	Allstate Insurance Co.	Office automation system abandoned after deployment, cost $130 million.
1993	London Ambulance Service	Dispatch system cancelled in 1990 at $11.25 million[3]; second attempt abandoned after deployment, cost $15 million[3].
1993	Greyhound Lines Inc.	Bus reservation system crashed repeatedly upon introduction and contributed to revenue loss of $61 million.
1992	Budget Rent-A-Car, Hilton-Hotels, Marriott Int'l,American Airlines	Travel reservation system cancelled after $165 million is spent.
1990	AT & T Wireless	Massive shutdown of their network [Neumann, 1990] due to a bug in the program code which is part of software upgrade to speed up calling function.
1985	Atomic Energy Commission Limited (AECL) and CGR	Subtle bug found in Therac-25 radiation therapy machine caused three persons dead and another three persons critically injured [Jackson, 2004, Savive, 2010].
1962	National Aeronautics and Space Administration (NASA)	During the launch of Mariner I rocket of the first Mariner mission, incorrect signals guidance (due to a missing superscript bar in a written formula) to the spacecraft resulted in off course.

[1] Converted to U.S. dollars using current exchange rates of press time.
[2] Converted to U.S. dollars using exchange rates for the year cited, according to the Statistical Abstract of United States, 1996.
[3] Converted to U.S. dollars using exchange rates of press time.

B

Spectra Metrics Surfaces

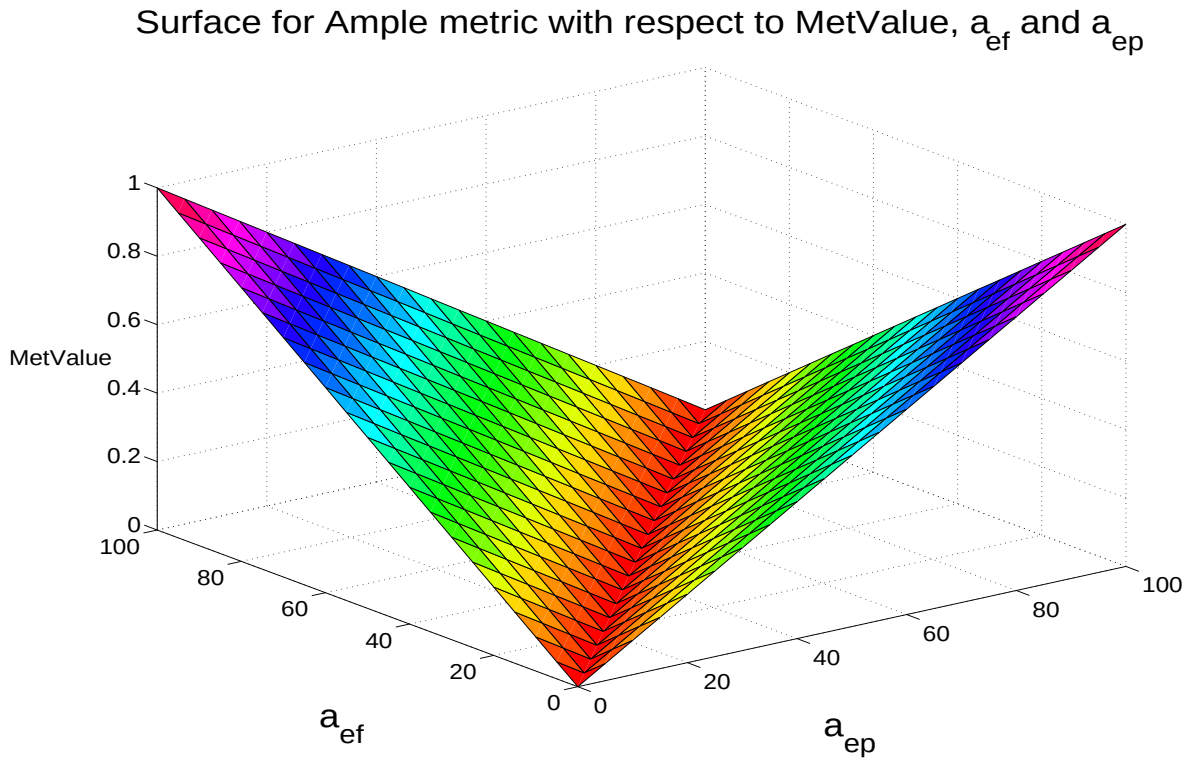

Figure B.1: Surface for Ample metric

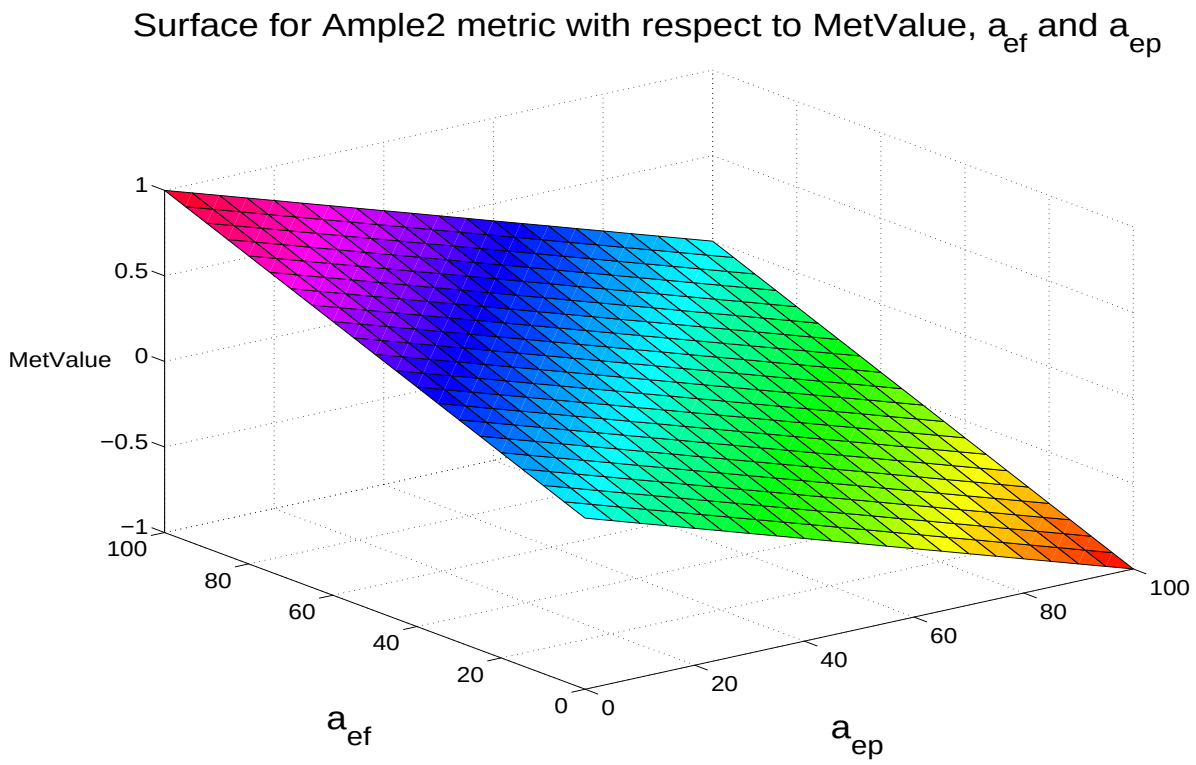

Figure B.2: Surface for Ample2 metric

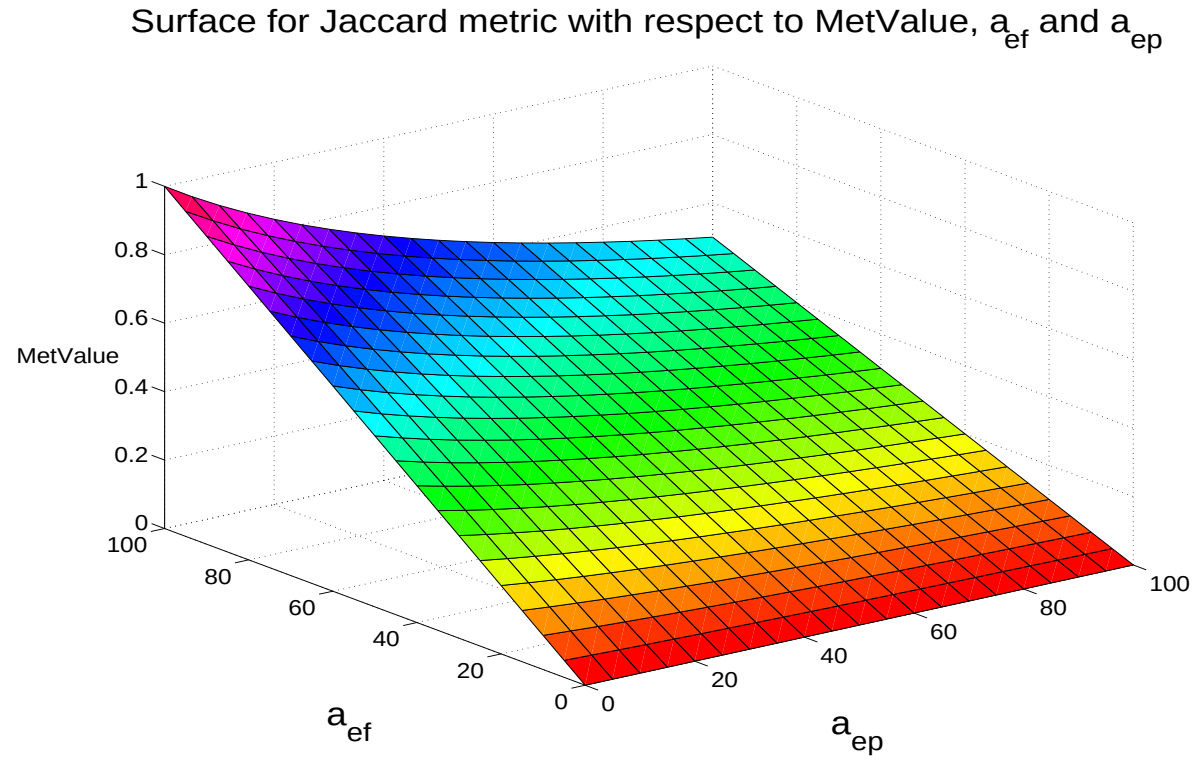

Figure B.3: Surface for Jaccard metric

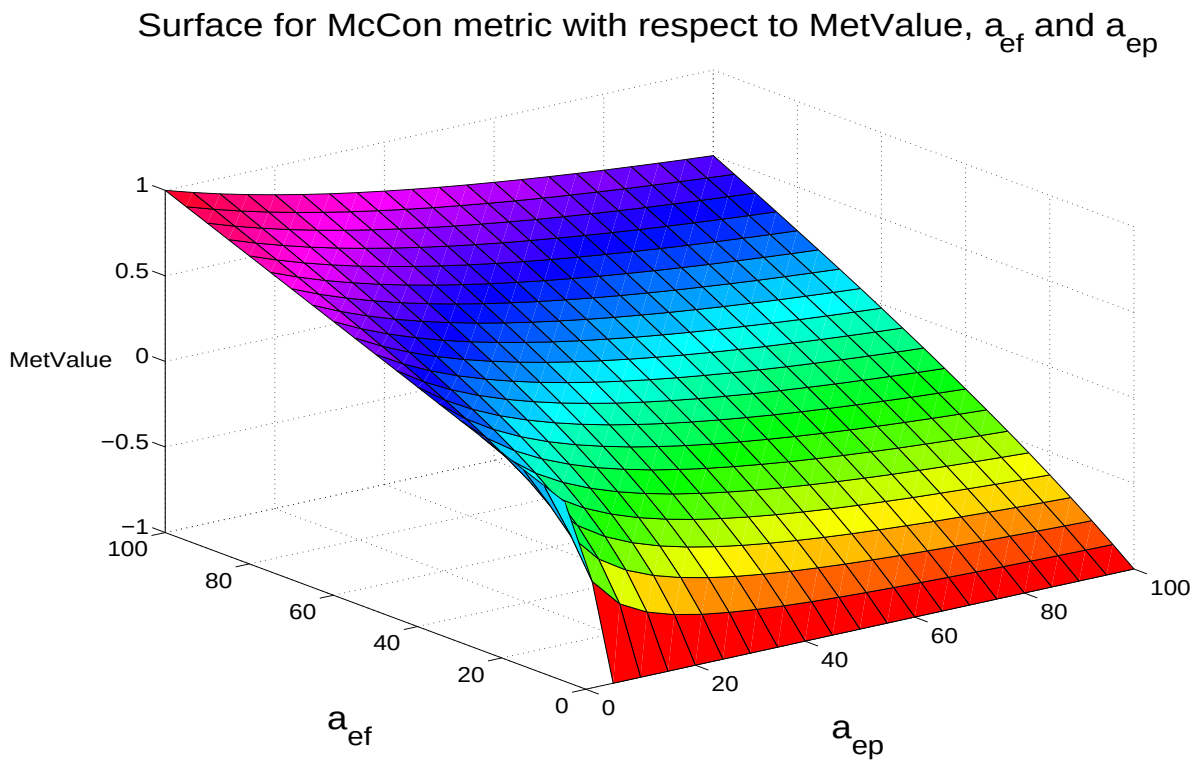

Figure B.4: Surface for McCon metric

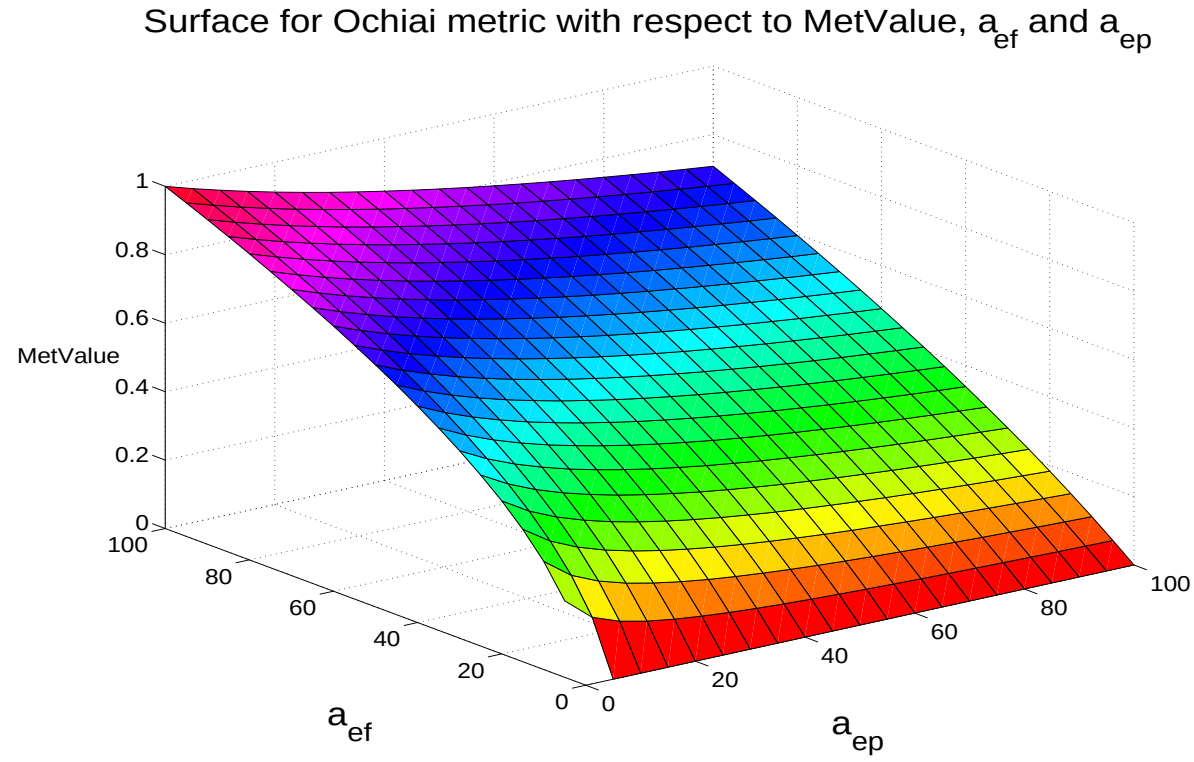

Figure B.5: Surface for Ochiai metric

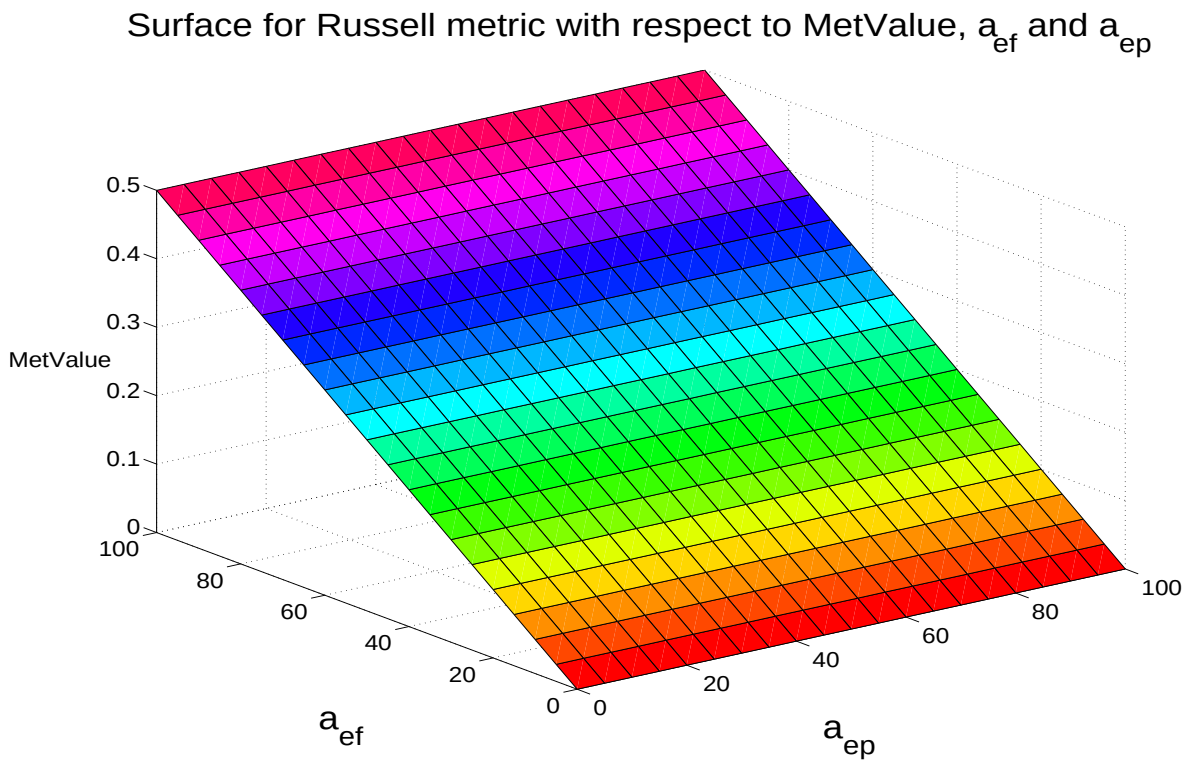

Figure B.6: Surface for Russell metric

C

Bug Localization Performance vs Bug Consistency, q_e for Respective Spectra Metrics

C.1 Single Bug Programs

In this section, we show the relationship of bug localization performance with respect to bug consistency, q_e, for other spectra metrics not shown in Chapter 6 on the Siemens Test Suite and the subset of the Unix Test Suite. We also detail our study of the latter relationship for the single bug programs of Concordance and Space.

C.1.1 Siemens Test Suite

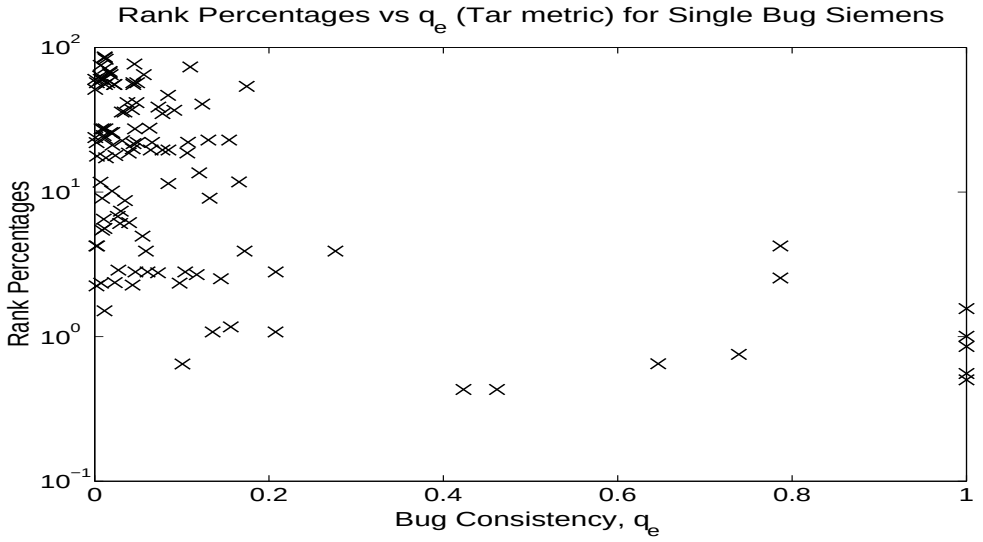

Figure C.1: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the Tarantula (Tar) metric

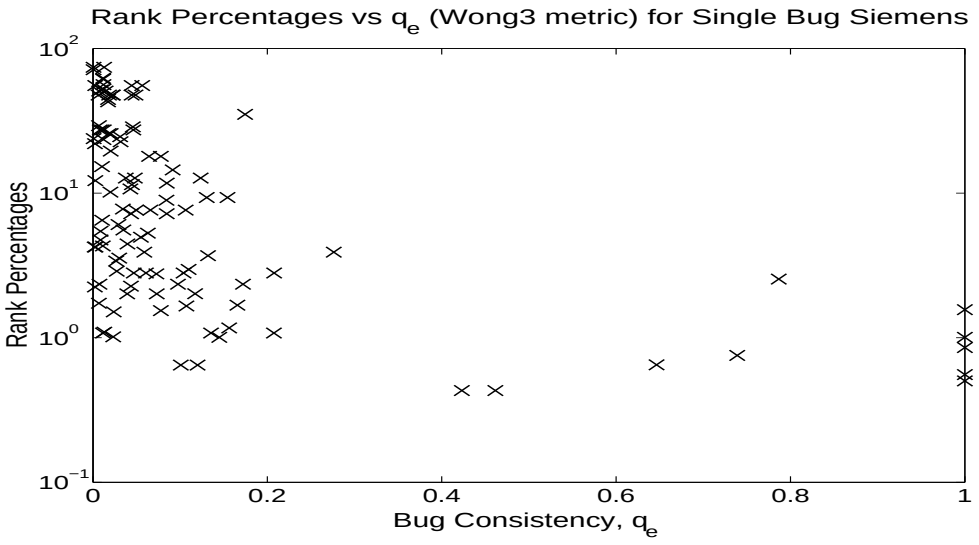

Figure C.2: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the Wong3 metric

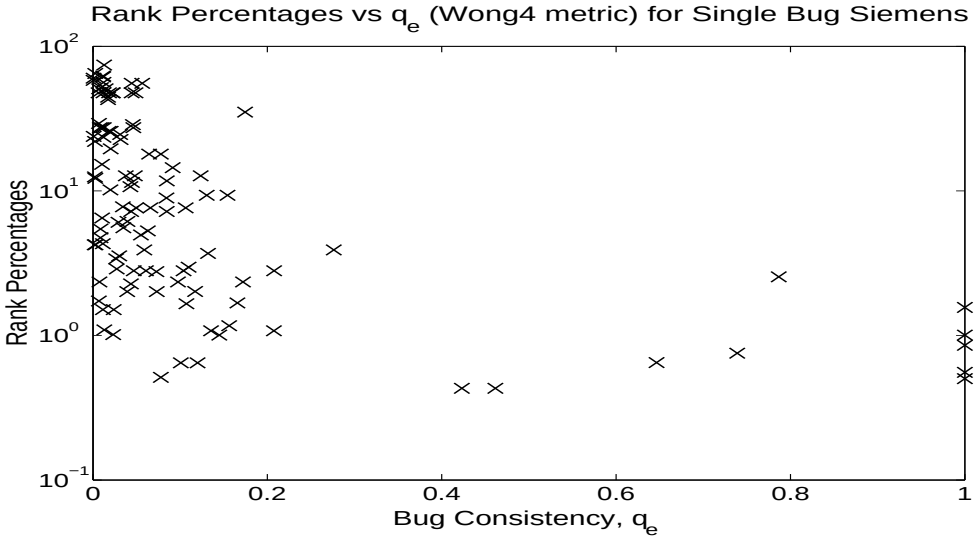

Figure C.3: Rank Percentages vs q_e for the Single Bug Siemens Test Suite with respect to the Wong4 metric

C.1.2 Subset of the Unix Test Suite

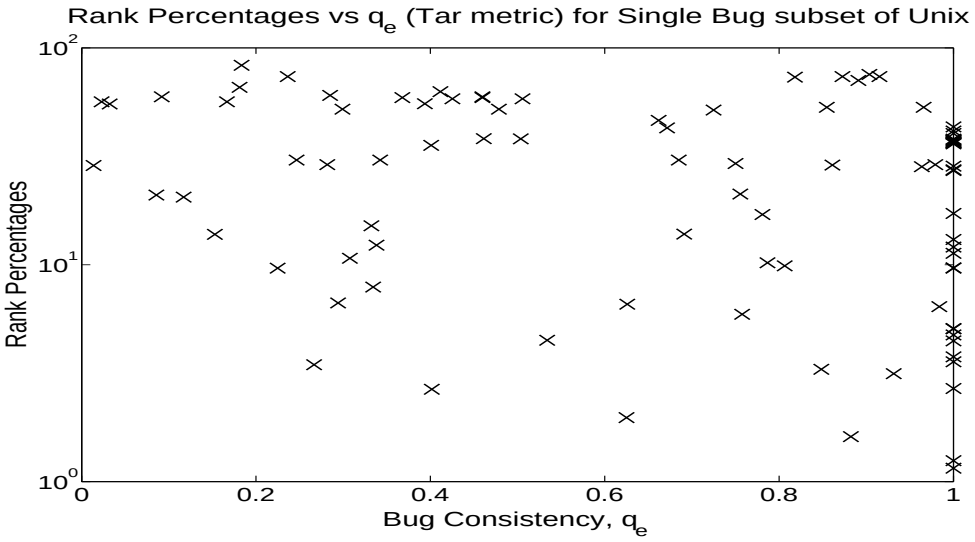

Figure C.4: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the Tarantula (Tar) metric

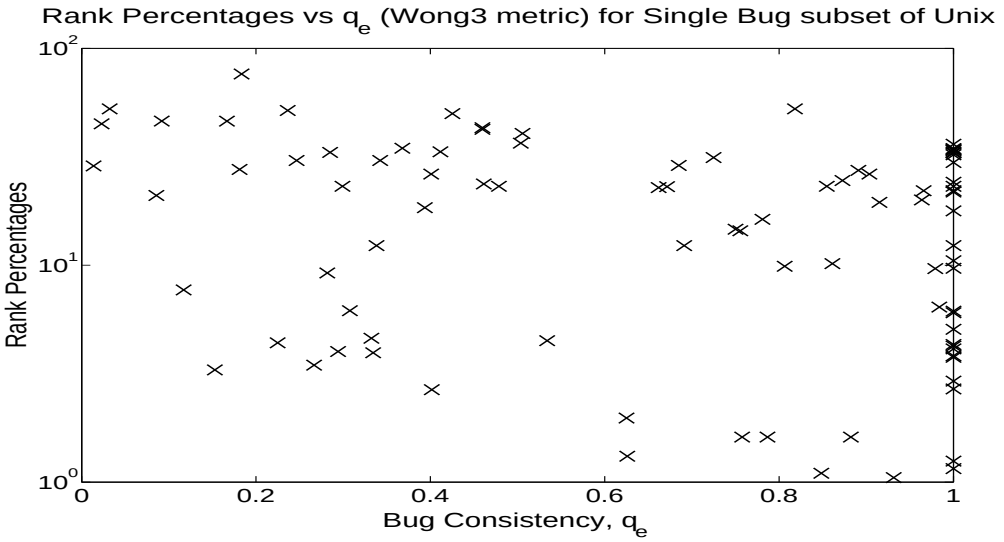

Figure C.5: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the Wong3 metric

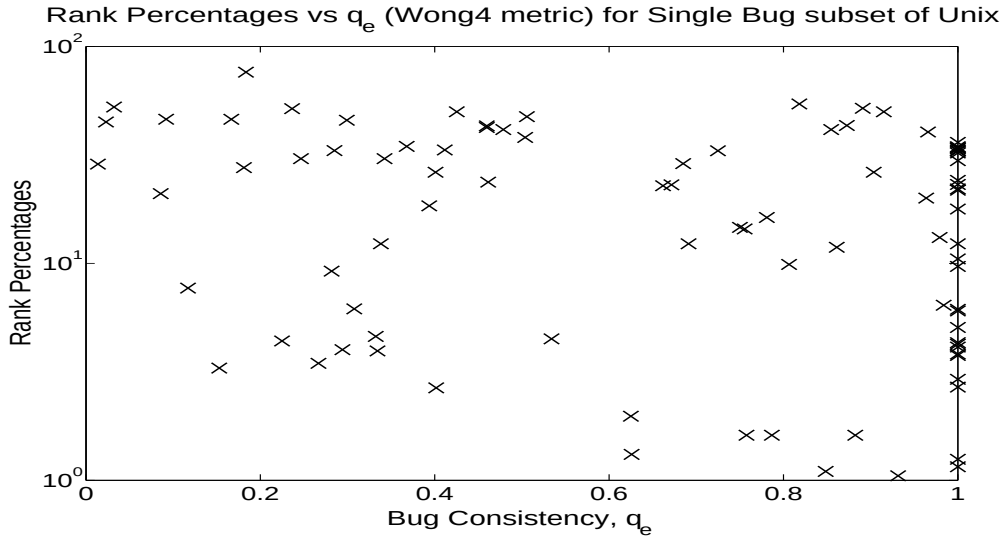

Figure C.6: Rank Percentages vs q_e for the Single Bug of subset of the Unix Test Suite with respect to the Wong4 metric

C.1.3 Concordance

In this section, we evaluate the relationship of bug localization performance with the bug consistency, q_e, for the Concordance using several spectra metrics. Each point in the below figures refers to the buggy statement of a typical program of the 11 single bug Concordance programs. We could not draw any strong conclusion of the points in Figure C.7 – Figure C.12 as there are only 11 programs in Concordance.

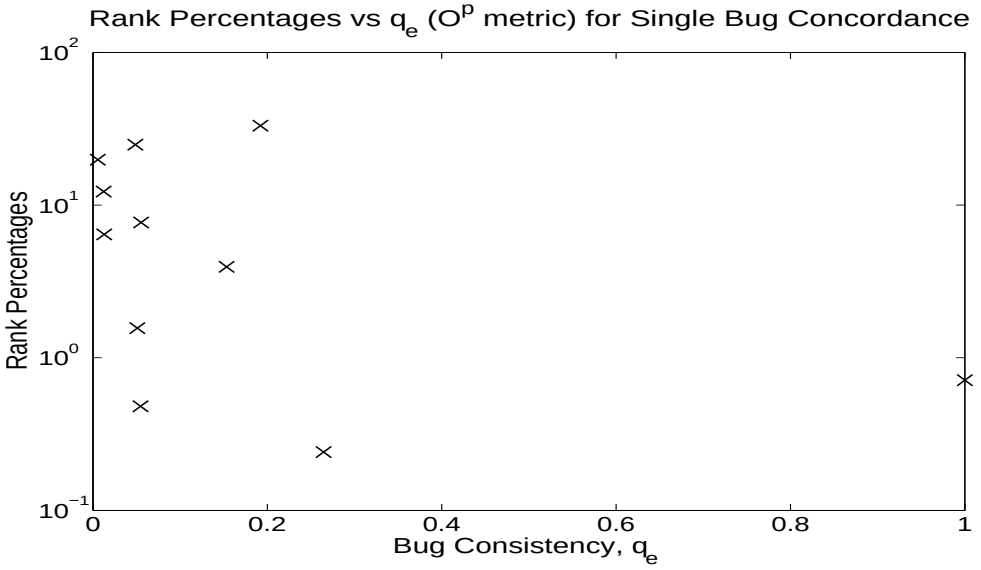

Figure C.7: Rank Percentages vs q_e for the Concordance with respect to the O^p metric

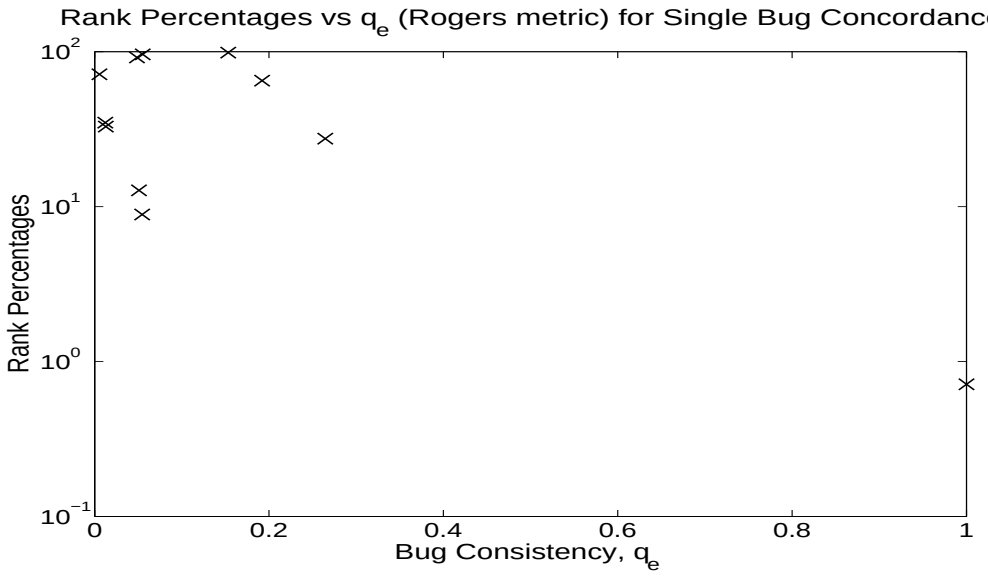

Figure C.8: Rank Percentages vs q_e for the Concordance with respect to the Rogers metric

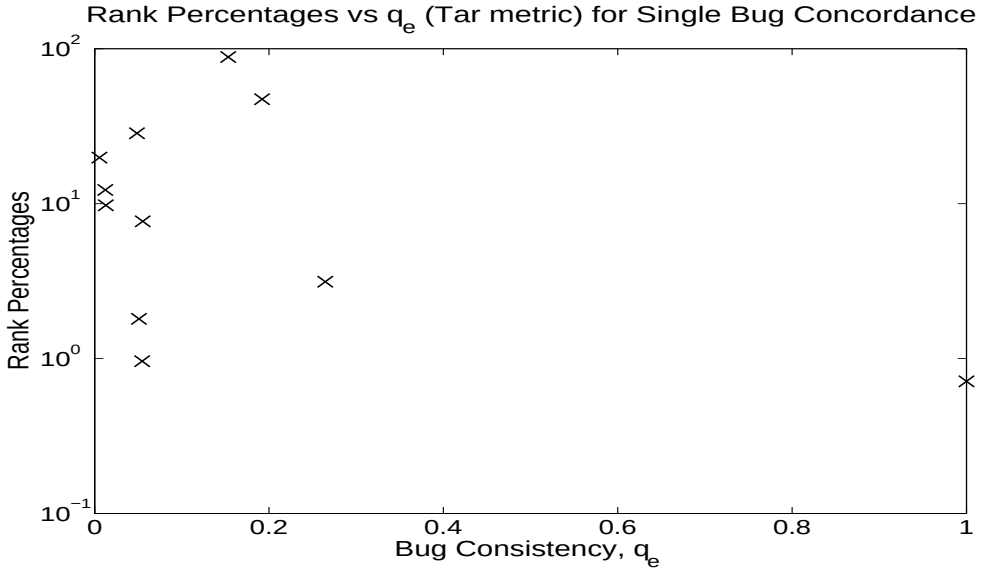

Figure C.9: Rank Percentages vs q_e for the Concordance with respect to the Tarantula (Tar) metric

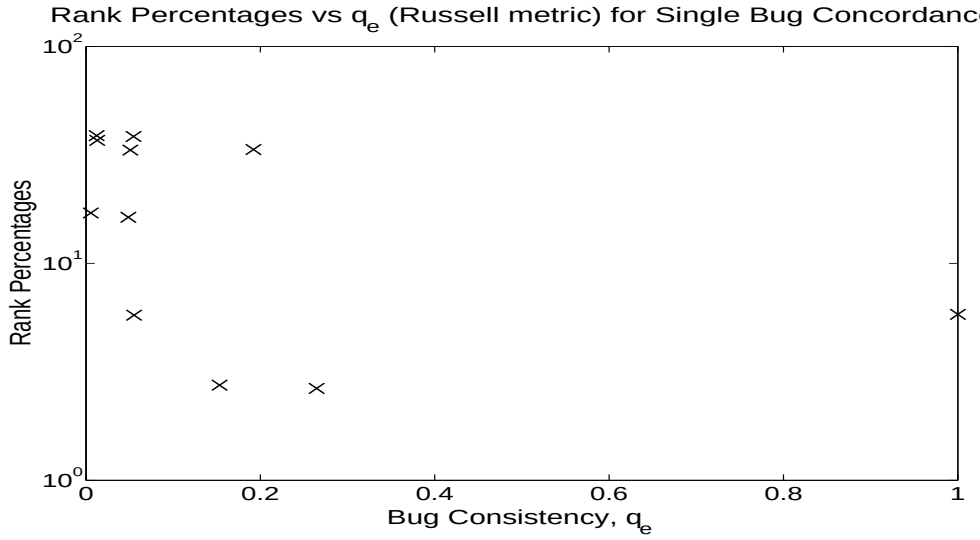

Figure C.10: Rank Percentages vs q_e for the Concordance with respect to the Russell metric

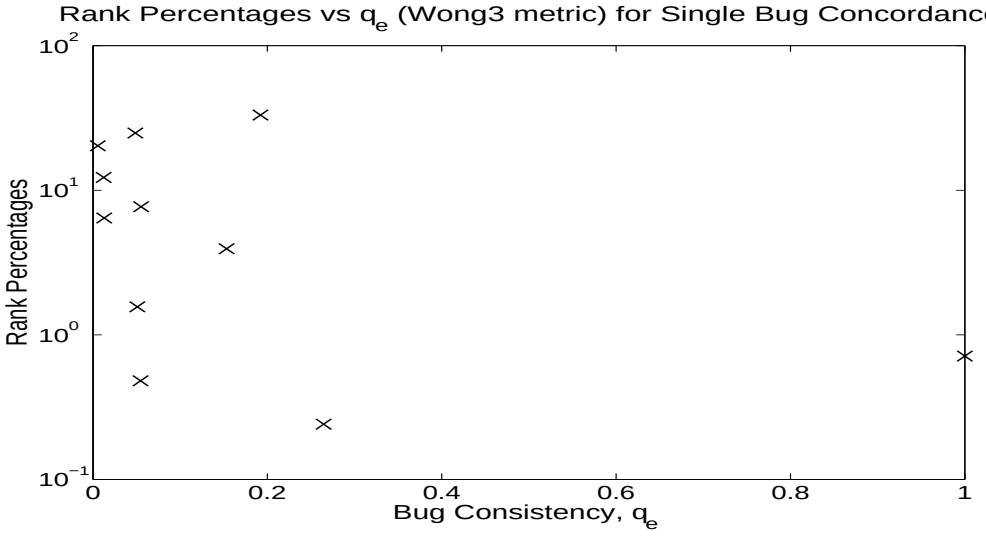

Figure C.11: Rank Percentages vs q_e for the Concordance with respect to the Wong3 metric

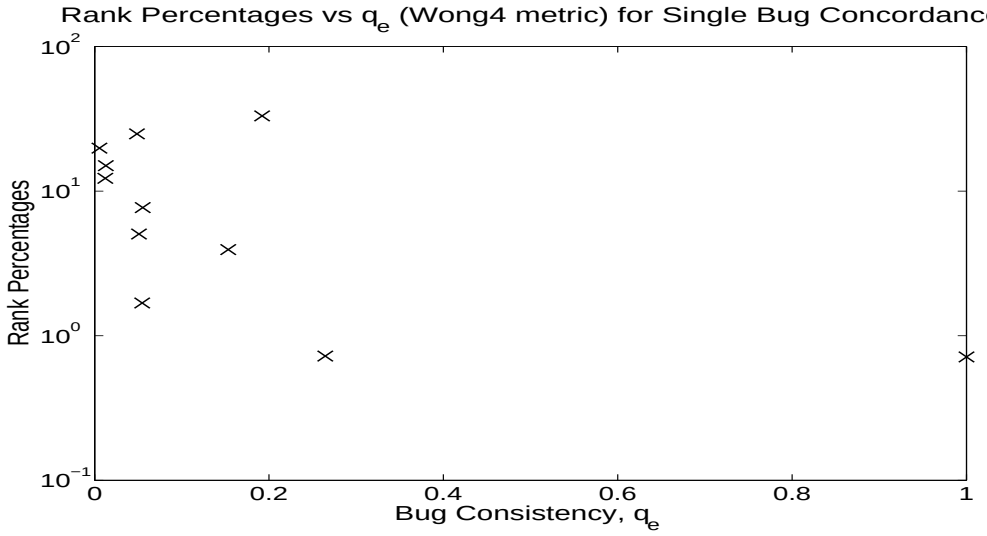

Figure C.12: Rank Percentages vs q_e for the Concordance with respect to the Wong4 metric

C.1.4 Space

We also study the relationship between bug localization performance and bug consistency, q_e, on the single bug programs of Space (15 programs) using the same set of spectra metrics (Subsection 6.3.1, Subsection 6.3.2, and Subsection C.1.3). We use the *Subset* of the Space test suite (randomly selected subset of Space test suite in 10 bins). Figure 5.13 of Chapter 5 shows that the bug localization performance is very similar in all the 10 bins. Therefore, we only choose to show the relationship of the bug localization performance with the bug consistency, q_e, for one of the 10 bins of Space *Subset* in this study.

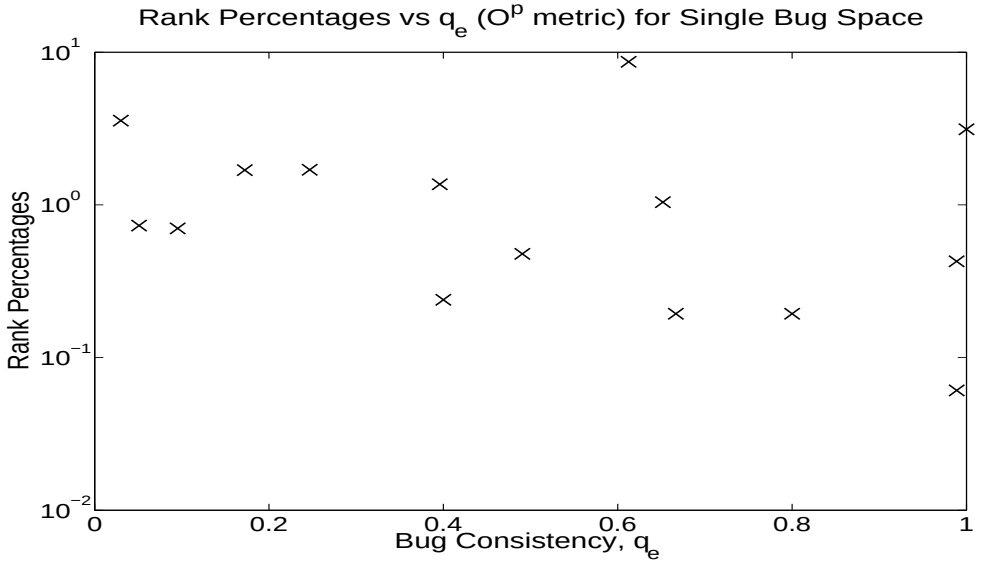

Figure C.13: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the O^p metric

O^p and Rogers metrics in Figure C.13 and Figure C.14 respectively show that the points of rank percentages are spread out as q_e increases. We also plot similar relationship

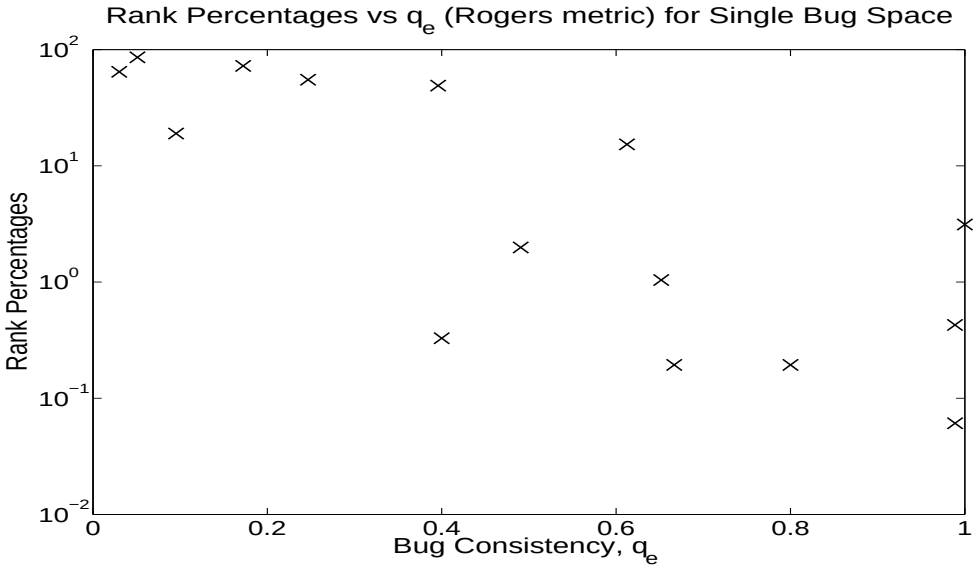

Figure C.14: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the Rogers metric

on Space for other metrics, namely, Tarantula, Russell, Wong3, and Wong4 in Figure C.15, Figure C.16, Figure C.17, and Figure C.18 respectively.

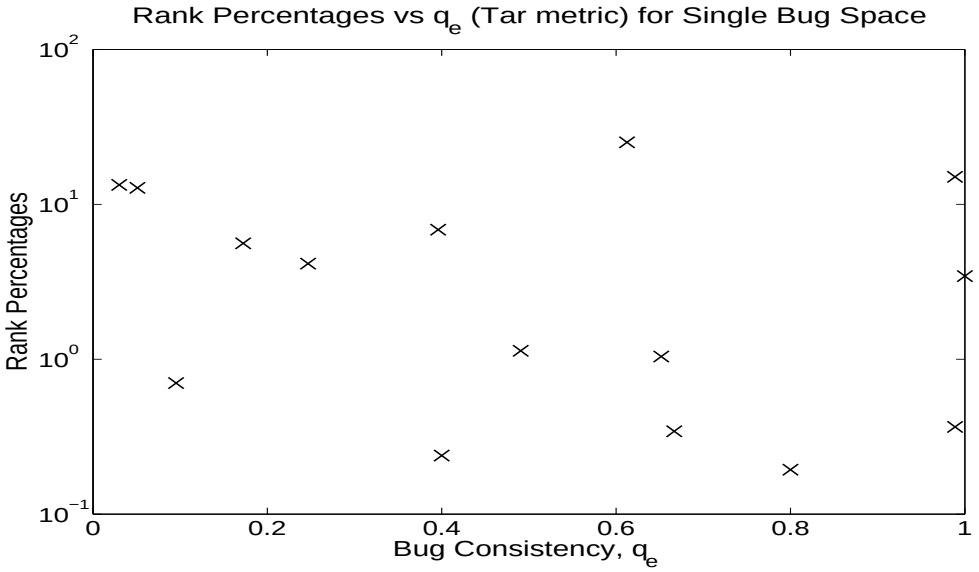

Figure C.15: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the Tarantula (Tar) metric

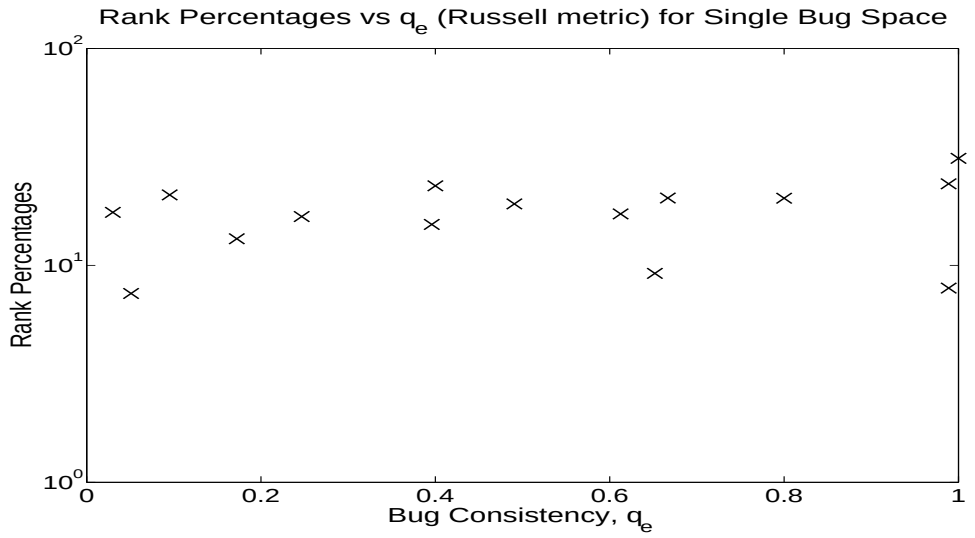

Figure C.16: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the Russell metric

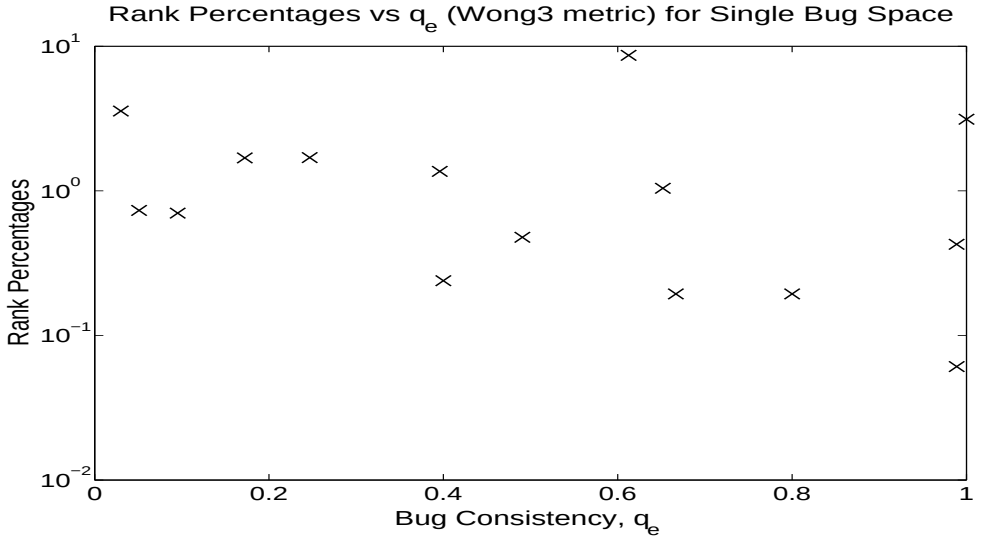

Figure C.17: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the Wong3 metric

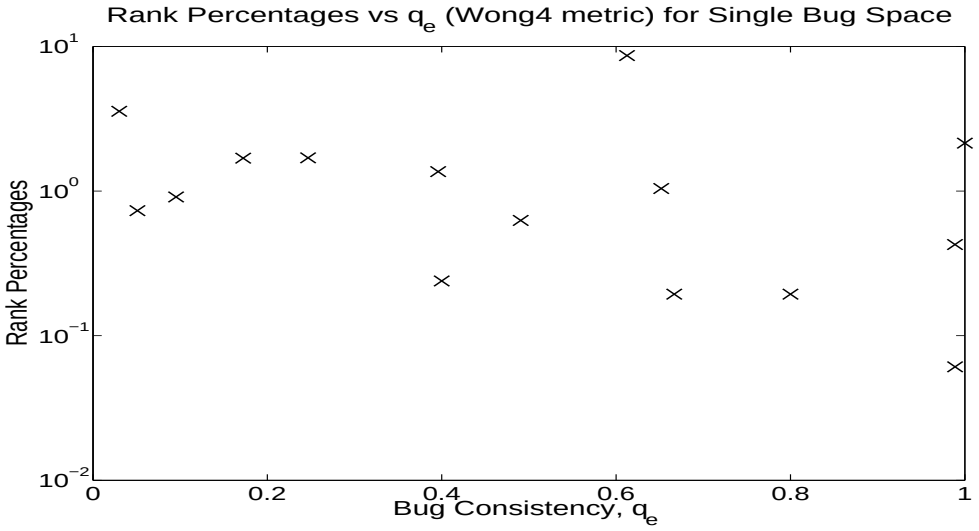

Figure C.18: Rank Percentages vs q_e for the Single Bug Space Programs with respect to the Wong4 metric

C.2 Multiple-bug Space Programs

In this section, we plot the relationship between the bug localization performance and bug consistency, q_e, for the 13 multiple-bug Space programs. Similarly to Subsection C.1.4, we only show the relationship of the bug localization performance with the bug consistency, q_e, for one of the 10 bins of the *Subset* of the Space test suite in this study.

In Table 5.17, we observe that Zoltar performs the best in bug localization performance of the multiple-bug Space programs as compared to the Tarantula (Tar) metric. We plot these two metrics on the 13 programs of the multiple-bug Space programs in Figure C.19 and Figure C.20 respectively. We observe the points of rank percentages in these figures are spread out as q_e increases.

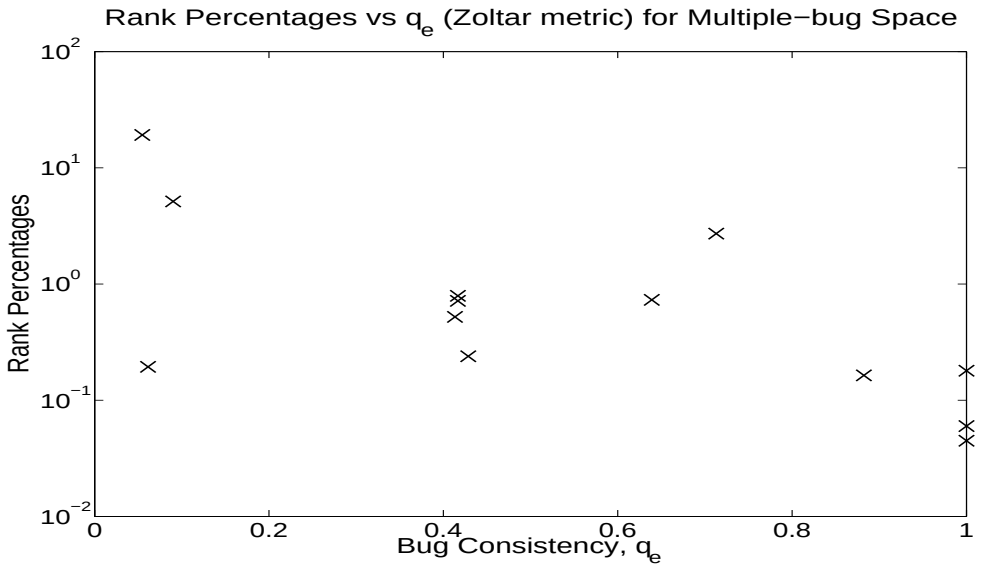

Figure C.19: Rank Percentages vs q_e for the Multiple-bug Space Programs with respect to the Zoltar metric

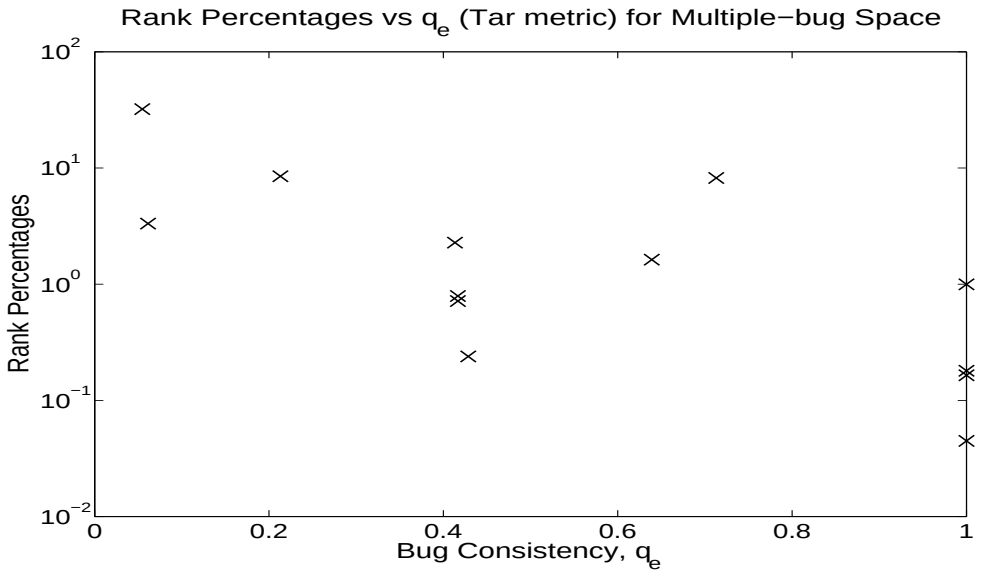

Figure C.20: Rank Percentages vs q_e for the Multiple-bug Space Programs with respect to the Tarantula (Tar) metric

D

Information of Unique (Non-redundant) and Redundant Test Cases

D.1 Single Bug Programs

Table D.1: Breakdown of Unique Test Cases (on average) for the Single Bug Siemens Test Suite, the subset of the Unix Test Suite, Concordance, and Space Programs

Program	Test Cases	Unique Test Cases	Unique Test Cases (%)
Cal	162	21	12.96
Checkeq	332	34	10.24
Col	156	48	30.77
Concordance	372	132	35.48
print_tokens	4130	1855	44.92
print_tokens2	4115	1735	42.16
replace	5542	2025	36.54
schedule	2650	469	17.69
schedule2	2710	664	24.50
Spline	700	83	11.86
tcas	1608	10	0.62
tot_info	1052	174	16.54
Tr	870	72	8.276
Uniq	431	122	28.31
Space *Subset*	1103	895	81.14

D.2 Multiple-bug Programs

Table D.2: Breakdown of Unique Test Cases (on average) for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite

Program	Test Cases	Unique Test Cases	Unique Test Cases (%)
Cal	162	20	12.34
Checkeq	332	29	8.735
Col	156	49	31.41
print_tokens2	4115	987	23.99
replace	5542	2029	36.61
schedule2	2710	667	24.61
Spline	700	82	11.71
tcas	1608	12	0.7463
tot_info	1052	177	16.82
Tr	870	77	8.85
Uniq	431	115	26.68

Table D.3: Breakdown of Unique Test Cases (on average) for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite

Program	Test Cases	Unique Test Cases	Unique Test Cases (%)
Cal	162	19	11.73
Checkeq	332	26	7.831
Col	156	49	31.41
print_tokens2	4115	994	24.16
replace	5542	2063	37.22
schedule2	2710	658	24.28
Spline	700	81	11.57
tot_info	1052	172	16.35
Tr	870	79	9.080
Uniq	431	117	27.15

Table D.4: Breakdown of Unique Test Cases (on average) for the Multiple-bug Space Programs

Program	Test Cases	Unique Test Cases	Unique Test Cases (%)
Space *Subset*	1225	1068	87.18

E

Varying the Number of Unique Test Cases with respect to Bug Localization Performance

In this appendix, we detail the study of varying the number of unique test cases for single bug and multiple-bug programs. In this evaluation, we vary the different percentages of the unique pass and fail test cases. For example, if we have 100 unique pass and fail test cases respectively, for 10%, 20%, 30%, 40%, 50%, and 80%, the effective unique pass and fail test cases we evaluate are 10, 20, 30, 40, 50, and 80 unique pass and fail test cases respectively. 100% simply refers to using all the unique pass and fail test cases. Our hypothesis is *the bug localization performance using all the unique test cases (100%) improves as compared to using only 10% of the unique pass and fail test cases*.

E.1 Single Bug Programs

We detail the study of varying the number of unique test cases for the single bug Concordance and Space programs. As we have 10 bins of Space programs for the *Subset* of the entire test suite of Space, we report the average of bug localization performance across these bins for different number of unique test cases.

Table E.1: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass and Fail Test Cases - Single Bug Concordance Programs

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
O,O^p	17.43	15.31	14.24	12.38	12.01	10.67	10.14	0.1540
Zoltar	17.55	15.35	14.26	12.39	12.01	10.67	10.14	0.1540
Kulczynski2	17.63	15.45	14.39	12.47	12.08	10.74	10.20	0.1540
McCon	17.63	15.45	14.39	12.47	12.08	10.74	10.20	0.1540
JacCube	17.50	15.44	14.40	12.54	12.14	10.77	10.22	0.1540
M2	17.62	15.53	14.54	12.68	12.21	10.84	10.40	0.1540
Ochiai	18.21	16.12	15.22	13.29	12.48	11.14	10.51	0.1540
Wong3	19.67	16.84	15.88	13.97	13.56	11.88	11.19	0.1540
Jaccard	18.81	17.30	16.75	15.78	14.69	12.73	12.35	0.1540
Wong4	22.80	18.40	16.32	13.26	13.14	12.24	12.75	0.1540

Continued on next page

Table E.1 – continued from previous page

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
AMean	19.41	18.50	17.45	16.85	16.96	15.15	13.76	0.1795
Pearson	19.41	18.51	17.50	16.93	17.08	15.17	13.91	0.1540
Ample2	19.47	18.70	17.76	17.13	17.39	15.64	14.68	0.2378
Tarantula	19.71	19.17	18.58	18.02	18.22	16.93	15.81	0.1540
Rogot2	25.52	22.97	24.09	21.92	22.49	19.93	19.73	0.3051
Overlap	26.42	24.56	23.72	22.94	23.36	21.72	21.03	0.0917
Russell	26.35	24.52	23.69	22.91	23.33	21.70	21.03	0.0917
Binary	26.35	24.52	23.69	22.91	23.33	21.70	21.03	0.0917
CBI Log	43.18	42.06	45.06	38.40	39.07	29.18	25.71	$<$0.05
Ample	46.30	44.17	43.56	42.36	40.92	40.27	40.06	0.3188

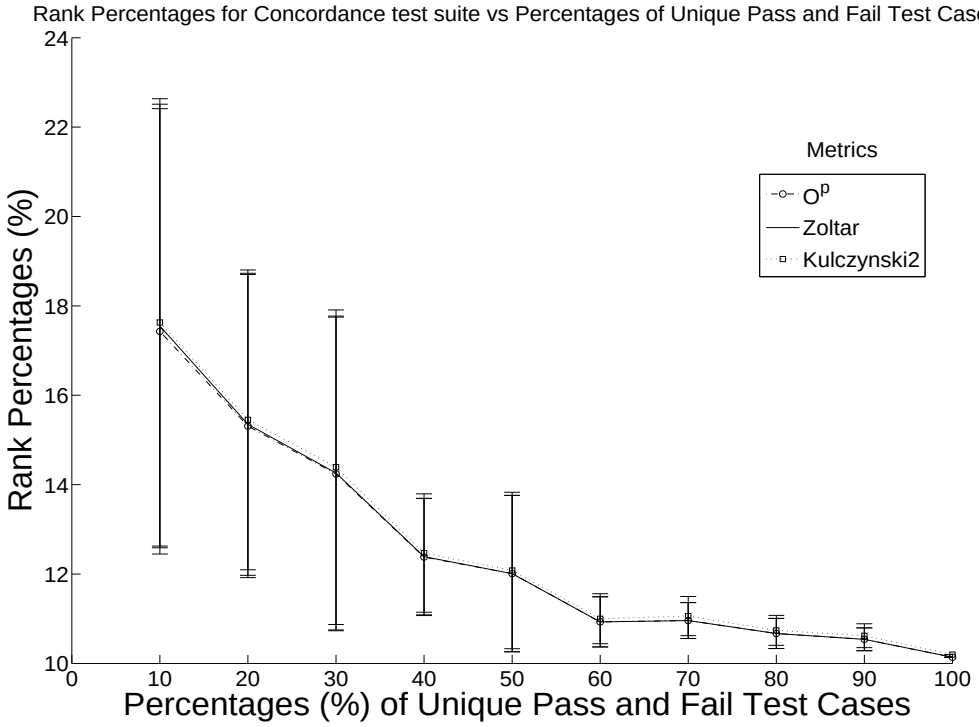

Figure E.1: Average Rank Percentages (on average) for the Single Bug Concordance Programs vs Percentages of the Unique Pass and Fail Test Cases

Table E.2: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass and Fail Test Cases - Single Bug Space Programs

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
O,O^p	3.12	2.18	1.95	1.83	1.75	1.66	1.63	<0.05
Wong3	4.60	2.96	2.63	2.43	2.30	1.99	1.74	<0.05
Zoltar	3.29	2.38	2.16	2.02	1.90	1.81	1.78	<0.05
JacCube	3.31	2.41	2.18	2.06	2.01	1.91	1.88	<0.05
M2	3.33	2.42	2.22	2.12	2.05	1.94	1.89	<0.05
Kulczynski2	3.47	2.59	2.38	2.25	2.14	2.05	2.01	<0.05
McCon	3.47	2.59	2.38	2.25	2.14	2.05	2.01	<0.05
Ochiai	3.73	2.97	2.72	2.56	2.44	2.31	2.23	<0.05
Wong4	5.15	3.44	3.59	2.56	2.67	2.60	2.54	<0.05
Rogot2	4.71	3.63	3.38	3.18	3.07	2.93	2.90	<0.05
Pearson	4.36	3.67	3.50	3.26	3.19	3.01	2.96	<0.05
AMean	4.34	3.71	3.62	3.40	3.38	3.18	3.15	<0.05
Ample2	4.46	3.69	3.55	3.39	3.33	3.22	3.15	<0.05
Jaccard	4.37	3.91	3.78	3.68	3.62	3.45	3.36	<0.05
Tarantula	6.70	6.71	6.81	6.71	6.67	6.54	6.42	<0.05
Ample	9.77	8.82	8.74	8.53	8.47	8.34	8.24	<0.05
CBI Log	23.80	15.00	14.03	14.19	14.42	11.91	8.46	<0.05
Russell	19.30	18.33	18.06	17.92	17.86	17.68	17.59	<0.05
Binary	19.30	18.33	18.06	17.92	17.86	17.68	17.59	<0.05
Overlap	20.25	19.37	19.07	18.85	18.73	18.45	18.31	<0.05

Table E.1 and Table E.2 show bug localization performance of Concordance and Space programs respectively for different number of unique pass and fail test cases. The bug localization performance for Column Unique (100%) of Table E.1 and Table E.2 are equivalent to the figures we obtained in Column Unique of Table 7.4 and Table 7.5 respectively.

In these tables, we observe improvement in bug localization performance for most of the metrics as we evaluate using larger number of unique pass and fail test cases. We observe the latter bug localization performances converge to its performance of bug localization using all the unique test cases (Column Unique (100%)). We report the p-value of our hypothesis and observe all the better performing metrics in the Table E.2 have p-value of less than 0.05. This indicates that, for Space, the improvement in bug localization performance for all the better performing metrics using all the unique test cases as compared to using only the 10% of the unique pass and fail test cases, is statistically significant with confidence greater than 95%. We also observe improvement in bug localization performance for Concordance in Table E.1. The latter improvement for better performing metrics is only significant with confidence of 84%.

In Figure E.1 and Figure E.2, we plot bug localization performance of better performing spectra metrics (O^p, Wong3, Zoltar, and Kulczynski2) using the randomly selected ten representative sets of unique test cases for different percentages of the unique pass and fail test cases. In these figures, we also plot the standard deviation (represented in error bars) of bug localization performance for the ten sets of randomly selected representative test

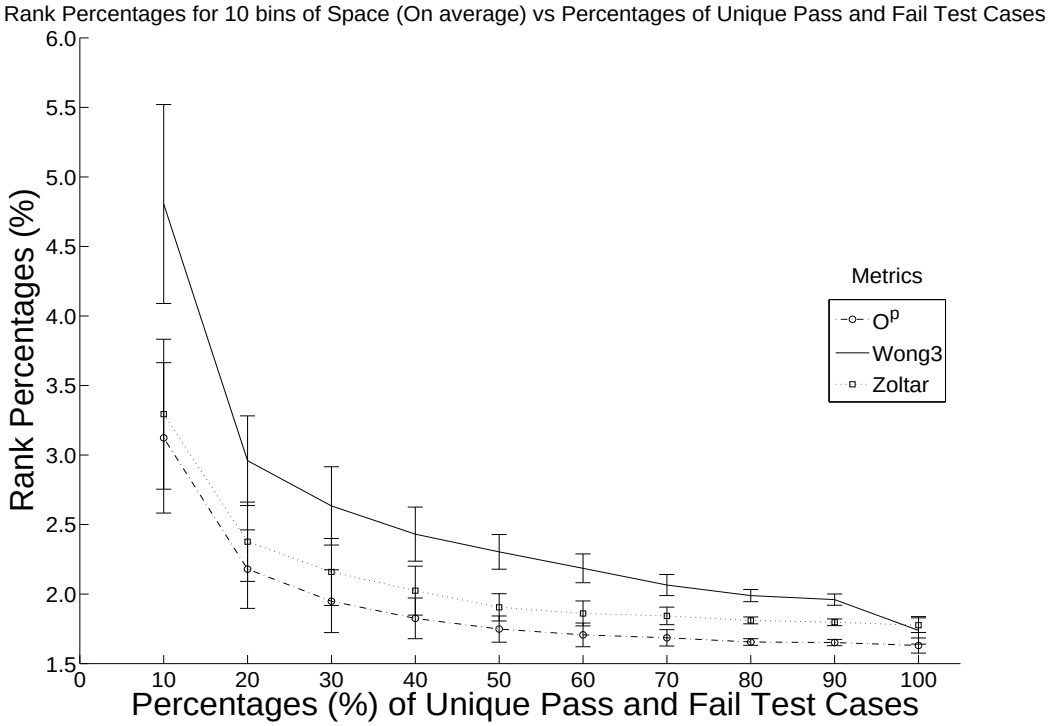

Figure E.2: Average Rank Percentages (on average) for the Single Bug Space Programs vs Percentages of the Unique Pass and Fail Test Cases

cases of unique pass and fail test cases. For Space programs, we take the average of the standard deviation of bug localization performance which we obtain across the 10 bins of the *Subset* of the entire test suite of single bug Space programs.

E.2 Multiple-bug Programs

We evaluate the two-bug and three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite in Table E.3 and Table E.4 respectively. We also report the p-value of our hypothesis in these tables.

Table E.3: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass and Fail Test Cases - Two-bug Programs Siemens Test Suite and the subset of the Unix Test Suite

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
Pearson	30.23	28.50	26.25	24.76	23.64	21.69	20.04	<0.05
Ample2	30.35	28.66	26.45	24.94	23.72	21.87	20.11	<0.05
AMean	30.26	28.54	26.30	24.81	23.71	21.65	20.13	<0.05
Jaccard	30.14	28.55	26.35	24.90	23.82	21.95	20.29	<0.05
Rogot2	30.81	29.00	26.68	25.24	24.05	21.97	20.40	<0.05
Wong4	32.03	29.79	27.24	25.25	23.97	21.83	20.58	<0.05
Kulczynski2	29.68	27.97	25.89	24.26	23.22	21.24	20.89	<0.05
McCon	29.68	27.97	25.89	24.26	23.22	21.24	20.89	<0.05
Ochiai	29.74	27.99	25.95	24.41	23.34	21.45	20.93	<0.05
CBI Log	35.10	33.31	33.01	31.96	29.65	25.80	21.13	<0.05
Zoltar	30.00	28.57	26.49	24.86	23.71	21.61	21.16	<0.05
Tarantula	31.03	29.78	27.82	26.44	25.40	23.56	21.61	<0.05
Wong3	30.10	28.85	27.45	26.38	25.66	23.60	22.33	<0.05
M2	29.96	28.42	26.54	25.03	23.96	22.40	22.35	<0.05
JacCube	30.02	27.34	25.37	24.52	23.27	22.03	22.37	<0.05
O^p	30.62	29.31	27.48	25.94	24.89	23.19	23.19	<0.05
Ample	33.83	32.26	30.33	28.83	27.52	25.70	23.67	<0.05
O	31.11	30.21	28.57	27.18	26.24	24.83	25.05	<0.05
Russell	37.30	36.06	35.43	34.86	34.26	33.15	32.44	<0.05
Binary	37.71	36.85	36.39	35.92	35.45	34.56	34.02	<0.05
Overlap	38.32	37.76	37.42	36.87	36.31	35.01	34.08	<0.05

Table E.4: Average Rank Percentages (on average) for the different Percentages Selection of the Unique Pass and Fail Test Cases - Three-bug Programs Siemens Test Suite and the subset of the Unix Test Suite

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
Wong4	30.43	26.76	25.41	24.39	23.57	22.67	22.29	<0.05
Ample2	29.17	25.96	24.67	23.91	23.48	23.02	22.86	<0.05
CBI Log	31.35	26.91	25.05	24.06	23.41	22.99	22.87	<0.05
Pearson	29.09	26.01	24.89	24.32	23.98	23.79	23.19	<0.05

Continued on next page

Table E.4 – continued from previous page

Metric	10%	20%	30%	40%	50%	80%	Unique (100%)	p-value
Rogot2	29.27	26.08	24.92	24.42	23.99	23.76	23.21	<0.05
AMean	29.11	26.04	24.90	24.35	24.00	23.75	23.28	<0.05
Ample	30.95	27.48	26.10	25.26	24.73	23.93	23.78	<0.05
Zoltar	29.56	27.07	26.14	25.83	25.38	25.21	25.04	<0.05
Kulczynski2	29.36	26.69	25.78	25.55	25.37	25.22	25.16	<0.05
McCon	29.36	26.69	25.78	25.55	25.27	25.22	25.16	<0.05
Ochiai	29.43	26.85	25.96	25.81	25.67	25.58	25.52	<0.05
Jaccard	29.62	27.11	26.23	26.05	25.86	25.77	25.73	<0.05
Wong3	29.60	27.14	26.88	26.84	26.80	26.21	26.21	<0.05
M2	29.77	27.47	26.89	26.88	26.84	26.79	26.79	<0.05
O^p	30.08	27.84	27.32	27.09	26.98	26.90	26.90	<0.05
Tarantula	30.52	29.00	28.46	27.88	27.62	27.42	26.94	<0.05
JacCube	30.24	28.43	27.95	27.90	27.52	27.31	27.03	<0.05
O	30.32	29.97	29.76	29.59	29.55	29.53	29.48	<0.05
Russell	35.11	32.35	31.42	30.94	30.61	30.17	30.05	<0.05
Binary	35.29	32.79	32.72	32.69	32.62	32.55	32.43	<0.05
Overlap	39.25	39.65	39.78	39.57	39.41	38.76	38.42	<0.05

In these tables, we observe that bug localization performance improves as we evaluate using larger number of unique pass and fail test cases for the Siemens Test Suite and the subset of the Unix Test Suite. For all the better performing metrics in these tables, we also observe the improvement in bug localization performance using all the unique test cases as compared to using only the 10% of the unique pass and fail test cases. The latter improvement is statistically significant with confidence greater than 95%.

For the two-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite, we plot bug localization performance for several better performing metrics (Pearson, Ample2, and Jaccard) for the ten sets of randomly selected representative unique pass and fail test cases (Figure E.3). Figure E.4 shows the plot of bug localization performance for several better performing spectra metrics (Wong4, Ample2, and Pearson) for the three-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite. These figures also similarly show that bug localization performance improves as we evaluate using larger number of the unique pass and fail test cases.

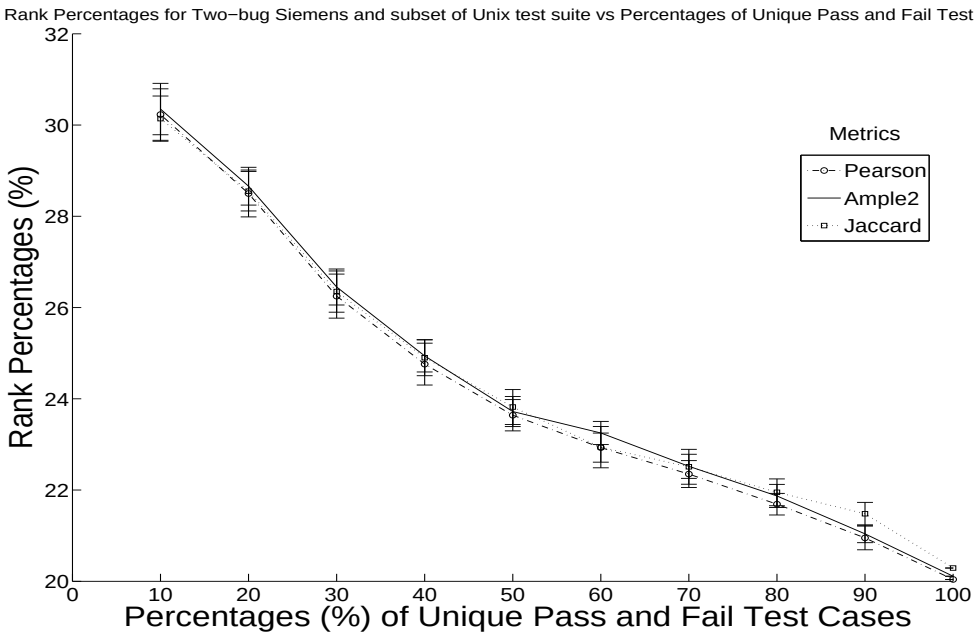

Figure E.3: Average Rank Percentages (on average) for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite vs Percentages of the Unique Pass and Fail Test Cases

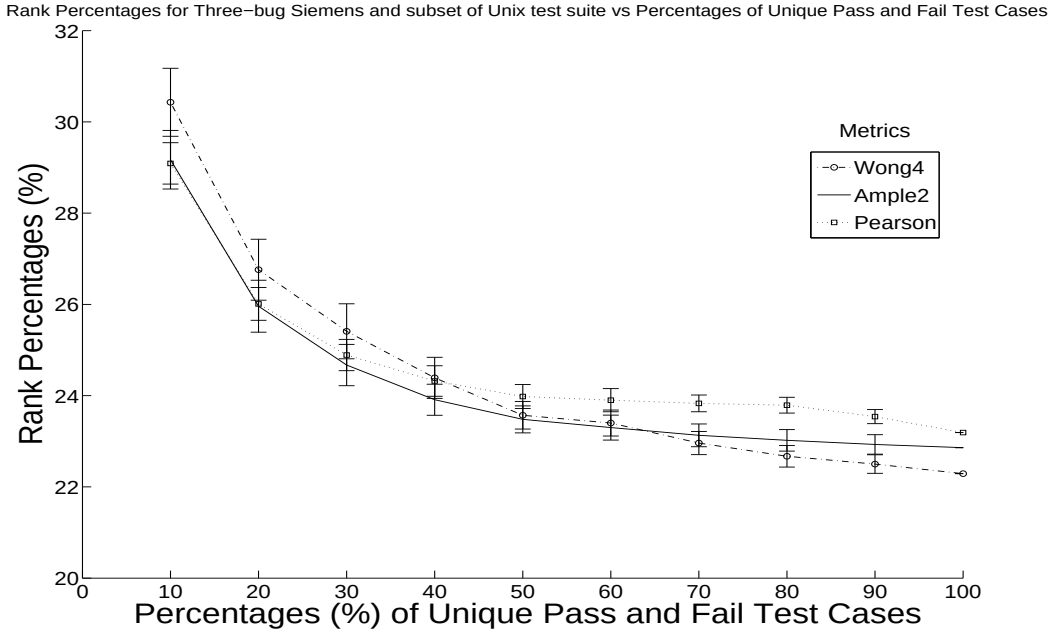

Figure E.4: Average Rank Percentages (on average) for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite vs Percentages of the Unique Pass and Fail Test Cases

F

Empirical Evaluation of the Proposed Incremental Ranking Approaches on Unique (Non-redundant) Test Cases

We used unique (non-redundant) test cases to evaluate bug localization performance in Chapter 7. Generally, we did not observe statistically significant improvement in bug localization performance by using unique test cases on better performing metrics. Some not so good metrics showed a slight improvement in bug localization performance using the unique (non-redundant) test cases. In this section, we are not interested in which spectra metric performs better in terms of bug localization performance. Instead, we want to investigate whether the proposed Weighted and incremental ranking approaches in Chapter 8 still perform robustly using the unique test cases. We use the term of non-redundant test cases and unique interchangeably.

We detail the evaluation of the Weighted and incremental ranking approaches on single bug and multiple-bug programs of our benchmarks in the following sections. These sections can be skipped as similar improvement of bug localization performance is observed using our proposed Weighted and incremental ranking approaches on the unique test cases of our benchmarks.

F.1 Single Bug Programs

Table F.1, Table F.2, and Table F.3 detail the evaluation of our proposed approaches using unique (non-redundant) test cases of the single bug programs, namely the Siemens Test Suite and the subset of the Unix Test Suite, Concordance, and Space programs respectively.

Table F.1: Average Rank Percentages for the Single Bug Siemens Test Suite and the subset of the Unix Test Suite (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	17.79	17.79	17.92	17.74	17.62	0.4597
O^p	17.79	17.80	17.90	17.90	17.78	0.5141
Zoltar	18.00	17.99	18.02	17.98	18.05	0.4485
Kulczynski2	18.56	18.50	18.49	18.52	18.34	0.4534

Continued on next page

Table F.1 – continued from previous page

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
McCon	18.56	18.50	18.65	18.70	18.77	0.5249
JacCube	18.67	18.57	18.52	18.60	18.36	0.4532
M2	18.88	18.79	18.67	18.68	18.81	0.4758
Wong3	18.90	18.00	17.91	18.04	17.91	0.2924
Ochiai	19.65	19.53	19.37	19.41	19.64	0.4483
Wong4	20.78	20.92	22.37	21.81	21.90	0.9880
Jaccard	21.58	21.03	21.26	21.10	20.94	0.4016
Pearson	22.19	21.59	21.56	21.86	21.73	0.4222
AMean	22.38	21.73	21.84	21.95	21.71	0.3694
Ample2	23.33	22.44	22.46	22.68	22.62	0.3772
Rogot2	23.57	22.72	22.36	22.60	22.86	0.4308
Tarantula	25.91	24.98	25.83	25.69	25.70	0.4487
CBI Log	28.11	25.77	27.08	28.03	27.94	0.3838
Binary	30.02	30.02	29.97	30.01	30.14	0.8704
Russell	30.02	30.02	30.19	30.24	30.14	0.4379
Ample	30.04	26.38	26.50	26.43	26.56	0.0663
Overlap	32.23	32.44	29.34	30.25	33.00	0.5214

Table F.2: Average Rank Percentages for the Single Bug Concordance (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	10.14	10.14	10.02	10.05	10.05	0.2417
O^p	10.14	10.14	10.18	10.15	10.12	0.2643
Zoltar	10.14	10.14	10.20	10.14	10.14	0.6369
Kulczynski2	10.20	10.14	10.12	10.25	10.07	0.0917
McCon	10.20	10.14	10.06	10.24	10.19	0.4721
JacCube	10.22	10.16	10.13	10.23	10.13	0.0705
M2	10.40	10.29	10.28	10.29	10.44	0.6880
Ochiai	10.51	10.62	10.55	10.78	10.66	0.5472
Wong3	11.19	10.14	10.18	10.09	10.14	0.1432
Jaccard	12.35	11.46	11.51	11.57	11.51	0.1716
Wong4	12.75	13.01	17.25	17.12	15.36	0.7296
AMean	13.76	11.44	11.40	11.51	11.37	0.1432
Pearson	13.91	11.49	11.48	11.50	11.49	0.1568
Ample2	14.68	11.88	11.79	11.67	11.82	0.0400
Tarantula	15.81	12.86	12.75	12.87	12.78	0.0220
Rogot2	19.73	14.39	13.51	13.48	13.57	0.1716
Binary	21.03	21.03	21.53	20.47	21.24	0.8635
Overlap	21.03	21.03	16.83	19.60	19.74	0.0337
Russell	21.03	21.03	20.50	20.19	21.35	0.6877
CBI Log	25.71	14.93	13.37	13.37	13.36	0.0547
Ample	40.06	27.45	26.53	25.75	25.89	0.0333

Table F.3: Average Rank Percentages (on average of 10 bins) for the Single Bug Space (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
O	1.63	1.63	1.63	1.63	1.57	0.4505
O^p	1.63	1.63	1.63	1.62	1.59	0.5165
Wong3	1.74	1.63	1.64	1.63	1.58	0.3544
Zoltar	1.78	1.77	1.79	1.79	1.72	0.5331
JacCube	1.88	1.87	1.88	1.89	1.82	0.5248
M2	1.89	1.90	1.92	1.89	1.83	0.5
Kulczynski2	2.01	2.01	2.01	2.02	1.95	0.4835
McCon	2.01	2.01	2.01	2.02	1.95	0.5165
Ochiai	2.23	2.22	2.22	2.24	2.13	0.5
Wong4	2.54	2.64	2.61	2.63	2.57	0.5
Rogot2	2.90	2.70	2.69	2.70	2.63	0.5331
Pearson	2.96	2.77	2.76	2.77	2.68	0.5495
AMean	3.15	2.97	2.96	2.96	2.89	0.4669
Ample2	3.15	2.94	2.92	2.92	2.83	0.4669
Jaccard	3.36	3.12	3.10	3.10	3.01	0.4669
Tarantula	6.42	6.13	6.14	6.15	6.00	0.4351
Ample	8.24	4.36	4.38	4.38	4.28	0.2807
CBI Log	8.46	6.40	6.15	6.06	5.90	0.1735
Binary	17.59	17.59	17.33	17.43	17.56	0.4854
Russell	17.59	17.59	17.34	17.63	17.61	0.7304
Overlap	18.31	18.22	16.99	19.01	18.72	0.5326

We observe a marginal drop in the effectiveness of bug localization performance by using the Weighted approach compared to using the Unweighted approach on O^p metric in Table F.1. However, we do not observe any difference in bug localization performance for the metric in the Concordance and Space programs (Table F.2 and Table F.3). In Table F.1 – Table F.3, we observe improvement in bug localization performance using the Weighted approach compared to the Unweighted approach for most metrics that are not located at the top of these tables. In these tables, most metrics show improvement in bug localization performance using the 10% Inc and 20% Inc approaches as compared to using the Unweighted approach. Better performing metrics such as O, O^p, and Kulczynski2 metrics only show the improvement in bug localization performance using the top-down incremental ranking approach (Incre.) as compared to the Unweighted approach for all the single bug programs. The improved average rank percentages using Incre. as compared to the Unweighted for O, O^p, and Kulczynski2 metrics are in the range from 0.06% to 0.17%, 0.01% to 0.04%, and 0.06% to 0.22% respectively.

In these tables, we also report the p-value of our hypothesis [Rice, 1989]. We establish a hypothesis; *the bug localization performance using our proposed top-down incremental ranking approach (Incre.) improves as compared to using the Unweighted approach*. We

apply the one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] to check the statistical significance of our hypothesis. We perform this statistical test only on one of the 10 bins in the Space program as the bug localization performances across the 10 bins are very similar (as shown in Figure 5.13 of Subsection 5.8.1). In all the unique test cases of the single bug programs, we do not observe any strong statistical significance of our hypothesis for all the metrics.

F.2 Multiple-bug Programs

We also evaluate our proposed approaches using the unique (non-redundant) test cases of the multiple-bug programs of the Siemens Test Suite, the subset of the Unix Test Suite, and Space programs. For Space programs, we use the *Subset* of the representative of the Space test suites (random test case selections in 10 bins). In this table, we report the average of bug localization performance for the Space programs across the 10 bins.

Table F.4: Average Rank Percentages for the Two-bug Siemens Test Suite and the subset of the Unix Test Suite (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Pearson	20.04	19.47	18.10	18.18	18.09	<0.05
Ample2	20.11	19.26	17.89	17.84	17.80	<0.05
AMean	20.13	19.59	18.21	18.32	18.07	<0.05
Jaccard	20.29	19.69	18.32	18.27	18.29	<0.05
Rogot2	20.40	20.26	19.10	19.08	19.10	<0.05
Wong4	20.59	20.63	20.63	19.81	19.48	<0.05
Kulczynski2	20.89	20.58	19.12	19.10	19.11	<0.05
McCon	20.89	20.58	19.11	19.11	19.03	<0.05
Ochiai	20.93	20.37	19.00	18.90	18.87	<0.05
CBI Log	21.13	20.51	19.46	19.49	19.51	<0.05
Zoltar	21.16	21.06	19.71	19.70	19.76	<0.05
Tarantula	21.61	20.82	19.63	19.59	19.64	<0.05
Wong3	22.33	22.05	20.49	20.46	20.51	<0.05
M2	22.35	21.68	20.25	20.30	20.25	<0.05
JacCube	22.37	21.84	20.52	20.41	20.41	<0.05
O^p	23.19	22.89	21.47	21.41	21.41	<0.05
Ample	23.67	23.75	22.45	22.56	22.65	<0.05
O	25.05	24.76	22.82	22.68	22.65	<0.05
Russell	32.44	32.02	26.77	26.73	26.47	<0.05
Binary	34.02	33.75	27.65	27.67	27.54	<0.05
Overlap	34.08	33.92	25.12	25.79	27.06	<0.05

Table F.5: Average Rank Percentages for the Three-bug Siemens Test Suite and the subset of the Unix Test Suite (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Wong4	22.29	21.37	18.28	17.88	17.70	<0.05
Ample2	22.86	22.41	18.44	18.35	18.39	<0.05
CBI Log	22.87	22.50	18.57	18.46	18.46	<0.05
Pearson	23.19	22.76	18.70	18.63	18.60	<0.05
Rogot2	23.21	23.06	18.98	19.03	19.00	<0.05
AMean	23.28	22.86	18.74	18.77	18.73	<0.05
Ample	23.78	23.73	19.83	19.77	19.70	<0.05
Zoltar	25.04	25.03	20.86	20.84	20.87	<0.05
Kulczynski2	25.16	24.50	19.61	19.56	19.47	<0.05
McCon	25.16	24.50	19.55	19.47	19.49	<0.05
Ochiai	25.52	24.84	19.78	19.82	19.83	<0.05
Jaccard	25.73	24.30	20.05	20.02	19.98	<0.05
Wong3	26.21	25.82	20.79	20.79	20.80	<0.05
M2	26.79	25.16	20.95	20.85	20.88	<0.05
O^p	26.90	25.60	21.40	21.31	21.32	<0.05
Tarantula	26.94	26.32	20.08	20.05	20.11	<0.05
JacCube	27.03	25.38	21.20	21.11	21.13	<0.05
O	29.48	29.08	22.96	22.93	22.87	<0.05
Russell	30.05	28.62	23.11	23.06	23.03	<0.05
Binary	32.43	32.05	24.39	24.45	24.42	<0.05
Overlap	38.42	38.35	24.25	24.45	24.78	<0.05

Table F.6: Average Rank Percentages (on average of 10 bins) for the Multiple-bug Space (Unique)

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
Zoltar	2.42	2.42	2.04	2.05	2.05	<0.05
JacCube	2.47	2.47	2.08	2.11	2.09	<0.05
O	2.55	2.55	2.16	2.16	2.17	<0.05
O^p	2.55	2.55	2.17	2.18	2.18	<0.05
Wong3	2.55	2.55	2.18	2.17	2.17	<0.05
M2	2.58	2.60	2.23	2.22	2.22	<0.05
Kulczynski2	2.63	2.61	2.24	2.25	2.25	<0.05
McCon	2.63	2.61	2.26	2.25	2.25	<0.05
Ochiai	2.77	2.76	2.39	2.39	2.39	<0.05
Wong4	2.80	2.75	2.36	2.38	2.37	<0.05
Jaccard	3.25	3.23	2.83	2.85	2.85	<0.05
Rogot2	3.26	3.22	2.84	2.83	2.83	<0.05
Pearson	3.41	3.38	2.98	2.98	2.99	<0.05
Ample2	3.51	3.44	3.04	3.02	3.02	<0.05

Continued on next page

Table F.6 – continued from previous page

Metric	Unweighted	Weighted	10% Inc	20% Inc	Incre.	p-value
AMean	3.59	3.52	3.13	3.13	3.13	<0.05
CBI Log	3.80	3.77	3.42	3.41	3.42	<0.05
Tarantula	4.36	4.31	3.94	3.94	3.95	<0.05
Ample	8.39	4.48	4.09	4.09	4.09	<0.05
Binary	17.85	17.85	10.50	10.52	10.52	<0.05
Russell	17.85	17.85	10.52	10.45	10.50	<0.05
Overlap	18.07	18.07	10.26	10.39	10.78	<0.05

Table F.4 and Table F.5 show that bug localization performance using the Weighted approach is better than the Unweighted approach for most of the metrics. In Table F.6, we do not observe much improvement in bug localization performance using the proposed Weighted approach compared to the Unweighted approach. In the evaluation of all the multiple-bug programs of our benchmarks (Table F.4, Table F.5, and Table F.6), the improvement of bug localization performance using 10% Inc, 20% Inc, and Incre. approaches as compared to the Unweighted approach are more obvious than the single bug programs (Table F.1, Table F.2, and Table F.3). In the former tables, the improved average rank percentages of the 10% Inc, 20% Inc, and Incre. approaches as compared to using the Unweighted approach for all the metrics range from 0.35% to 14.17%, 0.36% to 13.97%, and 0.36% to 13.64% respectively.

We also report the p-value of our hypothesis [Rice, 1989] in all the tables of the multiple-bug programs. We establish similar hypothesis of *the bug localization performance using our proposed top-down incremental ranking approach (Incre.) improves as compared to using the Unweighted approach*. We apply the one-sided Wilcoxon rank sum test [Hollander and Wolfe, 1973] to check the statistical significance of our hypothesis.

For the multiple-bug programs of the Siemens Test Suite and the subset of the Unix Test Suite, and Space (Table F.4, Table F.5, and Table F.6), we observe a statistically significant improvement in bug localization performance using the top-down incremental ranking approach (Incre.) as compared to the Unweighted approach with confidence greater than 95% for all the metrics.

G

Spectral Frequency Weighting Function on Concordance and Space programs

In this section, we detail the evaluation of our proposed frequency weighting function approach in Chapter 9 on other single bug programs such as Concordance and Space.

Table G.1: Average Rank Percentages for the Single Bug Concordance with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
O	50.37	50.69	53.66	49.29	51.37	52.61	47.95	49.21	**10.11**
O^p	23.41	24.09	21.12	16.16	11.52	**10.09**	**10.09**	10.12	10.11
Zoltar	20.56	24.05	20.91	21.06	17.88	17.90	12.96	10.12	**10.11**
Wong3	24.59	25.12	21.80	19.70	12.50	**9.83**	10.14	10.16	10.15
Kulczynski2	24.26	24.73	23.06	22.91	13.11	**10.20**	**10.20**	10.31	10.21
McCon	24.26	24.73	23.06	22.91	13.11	**10.20**	**10.20**	10.22	10.21
JacCube	18.18	17.71	14.55	13.27	**10.40**	10.42	10.42	10.43	10.43
M2	22.75	24.42	21.37	13.10	**10.94**	10.96	10.97	11.06	10.97
Ochiai	17.02	19.51	17.01	13.68	**11.18**	**11.18**	**11.18**	11.20	11.19
Wong4	24.33	28.13	23.00	24.15	27.07	28.54	28.85	22.91	**11.35**
Jaccard	**15.10**	15.51	17.09	17.57	17.76	17.76	17.77	17.69	17.68
Ample2	22.75	23.80	20.75	20.51	**18.02**	**18.02**	**18.02**	18.05	18.05
Pearson	21.48	23.63	21.29	20.73	18.40	18.40	**18.38**	18.43	18.42
AMean	18.31	23.08	21.40	20.77	**18.87**	**18.87**	**18.87**	18.89	18.90
Rogot2	22.84	23.41	21.01	20.71	**18.23**	18.30	18.30	18.32	19.24
Tarantula	19.73	**19.52**	19.94	19.83	20.01	20.01	20.01	20.03	20.03
Russell	28.37	29.09	25.81	25.18	25.26	24.39	24.55	24.15	**21.03**
Binary	50.37	50.69	53.66	49.59	51.44	52.50	48.86	50.17	**21.03**
Overlap	27.99	28.85	25.57	22.82	23.87	18.45	**18.03**	19.71	21.03
CBI Log	23.35	23.22	23.16	22.68	22.78	22.50	22.51	22.43	**22.35**
Ample	34.87	36.10	33.04	31.65	**27.50**	**27.50**	**27.50**	27.53	27.53

Table G.2: Average Rank Percentages (on average of 10 bins) for the Single Bug Space with respect to the Different α values

	α								
Metric	**0.1**	**0.5**	**1**	**2**	**4**	**8**	**10**	**20**	**Bin**
O	50.06	50.64	50.81	51.25	51.11	43.42	40.05	39.10	**1.64**
O^p	6.32	6.19	5.99	5.74	5.06	1.60	1.49	**1.48**	1.64
Wong4	7.50	8.38	7.93	7.24	6.75	3.70	3.73	3.08	**1.64**
Wong3	6.62	6.41	6.17	5.88	5.03	2.23	1.58	**1.55**	1.65
Zoltar	5.64	6.74	6.76	6.40	5.52	3.68	2.17	**1.65**	1.80
JacCube	4.82	5.22	4.67	3.87	1.98	**1.74**	**1.74**	**1.74**	1.90
M2	4.89	5.21	4.55	3.61	1.77	1.76	1.76	**1.75**	1.92
Kulczynski2	4.88	5.90	5.99	4.85	2.33	**1.91**	**1.91**	**1.91**	2.07
McCon	4.88	5.90	5.99	4.85	2.33	**1.91**	**1.91**	**1.91**	2.07
Ochiai	4.06	4.69	4.36	3.55	**2.10**	**2.10**	**2.10**	**2.10**	2.26
Rogot2	3.72	4.06	3.49	2.87	2.53	**2.51**	**2.51**	**2.51**	2.67
Ample2	2.58	2.59	2.57	**2.50**	**2.50**	2.51	2.51	2.52	2.68
Pearson	3.50	3.60	3.29	2.76	2.59	**2.56**	**2.56**	**2.56**	2.72
AMean	3.68	3.62	3.29	2.92	2.78	**2.77**	**2.77**	**2.77**	2.93
Jaccard	4.08	4.47	4.37	4.24	**2.99**	3.02	3.02	3.02	3.18
Tarantula	6.30	6.22	**6.17**	6.22	6.26	6.28	6.28	6.31	6.31
Ample	6.59	6.44	6.51	6.45	**6.39**	6.40	6.40	6.40	6.56
CBI Log	6.46	6.48	**6.42**	6.46	6.46	6.44	6.44	6.63	6.65
Russell	12.43	12.61	12.59	12.38	12.11	**12.02**	12.08	13.11	17.59
Binary	50.06	50.81	51.25	51.12	43.99	43.80	40.55	39.96	**17.59**
Overlap	**12.24**	13.62	13.97	12.80	12.59	18.83	18.72	20.91	18.31

Evaluation is performed using our proposed approach of frequency weighting function on Concordance which consists of only 11 single bug programs. In Table G.1, there is a marginal improvement in bug localization performance for the O^p, Kulczynski2, and Ochiai metrics using the proposed approach with the α value of 10 (average rank percentages of 10.09%, 10.20%, and 11.18%) as compared to using the traditional binary weighting approach (Bin) (average rank percentages of 10.11%, 10.21%, and 11.19%) respectively. Other metrics such as O, Zoltar, Wong4, Russell, Binary, and CBI Log do not show any improvement in bug localization performance using our proposed approach. In this table, we observe most metrics show the best improvement in bug localization performance by using our proposed approach with the α values ranging from 4 to 10.

We evaluate our proposed approach on the single bug programs of Space which consists of 15 programs. For the Space programs, we use the *Subset* of existing Space test cases (in 10 bins). In order to avoid the bias of bug localization performance on one particular bin, we take the average bug localization performance of the 10 bins of Space. The latter also applies when we evaluate bug localization performance for the traditional binary weighting approach (Bin). Table G.2 shows (on average) bug localization performance with respect to the different α values in Space. For each α variant we evaluate, we report the average of the bug localization performance of the *Subset* of Space programs

across the 10 bins. We observe that the O^p metric performs the best using our proposed approach with the α value of 20 when compared to the traditional binary weighting approach (Bin) with the average rank percentages of 1.48% and 1.64% respectively.

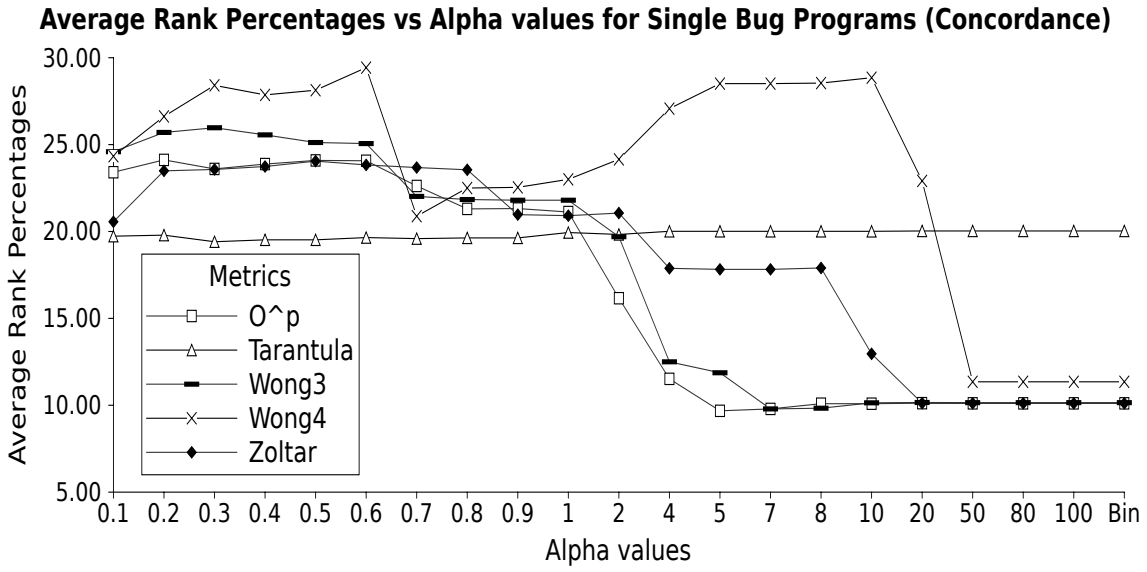

Figure G.1: Average Rank Percentages for the Different α values of the Single Bug Concordance

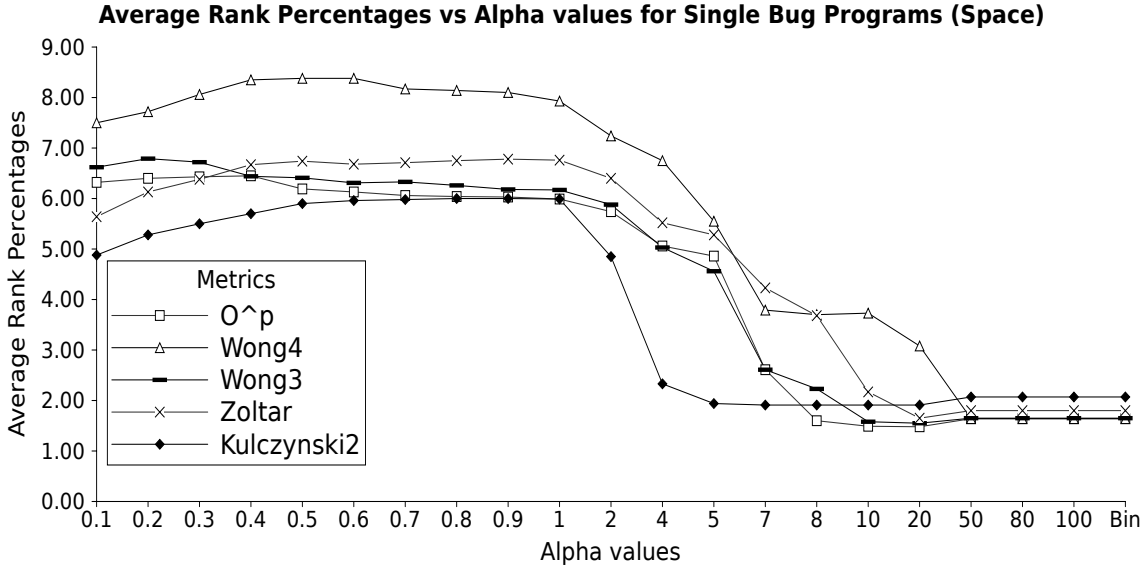

Figure G.2: Average Rank Percentages for the Different α values of the Single Bug Space

To see the trend of the bug localization performance on the choices of α values for both Concordance and Space programs, we plot the average rank percentages against different Alpha, α values (ranges from 0.1 to 100) including the traditional binary weighting approach (Bin) for several metrics in Figure G.1 and Figure G.2. For Concordance in Figure G.1, we observe that our proposed approach on the O^p and Tarantula metrics show improvement in bug localization performance in the range of the α value between 5 to 20

as compared to using the traditional binary weighting approach. In Figure G.2 for Space, all the metrics except Zoltar and Wong4 show slight improvement in bug localization performance with the range of the α value from 10 to 20 as compared to using the traditional binary weighting approach (Bin).

H

Bug Information of Respective Datasets

Table H.1: Table of Print_Tokens Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`case 16:ch=get_char( tstream_ptr->ch_stream );`	`case 16 :`	missing switch case statement
		`case 32 : ch=get_char (tstream_ptr-> ch_stream);`	added switch case statement as part of the new switch case
	`case 25 :`	`case 25 : token_ptr ->token_id=special( next_st);`	added switch case statement
	`case 32 : token_ptr ->token_id=special( next_st);`		removed switch case statement
		`case 32: return( EQUALGREATER);`	added switch case statement
	`case 32: return( EQUALGREATER);`		removed switch case statement
2		`case 12 :`	added empty switch case

Continued on next page

Table H.1 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
3	unget_char(ch, tstream_ptr->ch_stream);</br>	/*unget_char(ch, tstream_ptr->ch_stream);*/	commented line
5	token_ind=next_st=0;	next_st=0;	commented line
7	if(token_ind >= 80) break;	if(token_ind >= 10) break;	logic error

Table H.2: Table of Print_Tokens2 Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	if(id==0 && ch==59){ch =unget_char(ch,tp); if (ch==EOF)unget_error(tp);return(buffer); }		removed if conditions
2	if(ch==EOF)unget_error (tp);	unget_error(tp);	commented line
3	return(buffer);		removed return statement
4	if(ch==59)id=2;	if(ch==59)id=0;	logic error
5	return(FALSE);	return(TRUE);	logic error
6	if(isdigit(*(str+i)))	if(isdigit(*(str + i +1)))	logic error
7	{if(ch=='\n')	{if(ch=='\n'||ch==' ')	logic error
8	if(ch ==' ' || ch=='\ n' || ch==59)return(TRUE);	if(ch ==' ' || ch=='\n ' || ch==59 || ch == '\t')return(TRUE);	logic error
9	{ if(ch=='\n')	{ if(ch=='\n' || ch == '\t')	logic error
10	{ while (*(str+i) !='\0')	{ while (*(str)!='\0')	logic error

Table H.3: Table of Replace Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	if (src[*i - 1] == ESCAPE){	if (src[*i] == ESCAPE) {	logic error
2	if (src[*i - 1] == ESCAPE){	/* if (src[*i - 1] == ESCAPE){	commented line
	} else	} else*/	commented line

Continued on next page

Table H.3 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
3	`if ((m >= 0)&& (lastm != m)){`	`if ((m >= 0)){`	logic error
4	`if ((m >= 0)&& (lastm != m)){`	`if ((m >= 0)&& (i != m )){`	logic error
5	`for (k = src[*i-1]+1; k<=src[*i+1]; k++)`	`for (k = src[*i-1]+1; k< src[*i+1]; k++)`	logic error
6	`while ((i > offset))`	`while ((i >= offset))`	logic error
7	`return (c ==BOL || c ==EOL || c ==CLOSURE);`	`return (c ==BOL || c ==ANY || c ==CLOSURE);`	logic error
8	`return (c ==BOL || c ==EOL ||c ==CLOSURE);`	`return (c ==BOL || c ==EOL);`	logic error
9	`else if ((isalnum(src [*i - 1]))&&(isalnum( src[*i + 1]))&&(src[*i - 1] <= src[*i + 1]))`	`else if ((isalnum(src [*i - 1]))&& (isalnum( src[*i + 1])))`	logic error
10	`else if ((isalnum(src [*i - 1]))&& (isalnum( src[*i + 1]))`	`else if ((isalnum(src [*i - 1]))`	logic error
11	`&& (src[*i - 1] <= src [*i + 1]))`	`&& (src[*i - 1] > src [*i]))`	logic error
12	`#define MAXPAT MAXSTR`	`#define MAXPAT 50`	#define constant mutation
13	`i = i + 1;`	`if (m == -1)i = i + 1; else i = i + 2;`	added code
14	`if ((lin[*i] != NEWLINE)&& (!locate( lin[*i], pat, j+1)))`	`if ((lin[*i] != NEWLINE))`	logic error
15	`result = i;`	`result = i + 1;`	logic error
16	`return (c == BOL ||c ==EOL ||c ==CLOSURE);`	`return (c == BOL || c == EOL || c == CLOSURE || c == ANY);`	logic error
17	`result = ESCAPE;`	`result = NEWLINE;`	logic error
18	`if ((lin[*i] != NEWLINE)&& (!locate( lin[*i], pat, j+1)))`	`if ((!locate(lin[*i], pat, j+1)))`	logic error
19	`result = fgets(s, maxsize, stdin);`	`if (!fgets(s, 104, stdin)){*result = 0;}`	added code
20	`result = ESCAPE;`	`result = ENDSTR;`	logic error

Continued on next page

Table H.3 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
21	#define MAXPAT MAXSTR	#define MAXPAT 99	#define constant mutation
	result = fgets(s, maxsize, stdin);	result = fgets(s, maxsize - 1, stdin);	logic error
	if(*j >= maxset)	if(*j > maxset)	logic error
	else if((arg[i] == EOL)&&(arg[i+1]==delim))	else if((arg[i] == EOL))	logic error
22	if(arg[*i] == NEGATE)	if(arg[*i + 1] == NEGATE)	logic error
23	if(s[*i + 1]==ENDSTR)	if(s[*i] == ENDSTR)	logic error
24	if(lin[*i] == NEWLINE) advance = 0;	advance = 0;	removed if condition
25	if(lin[*i] == NEWLINE)	if(lin[*i] <= NEWLINE)	logic error
26	if((lin[*i] != NEWLINE)&& (!locate(lin[*i], pat, j+1)))	if((lin[*i] != NEWLINE)&& (!locate(lin[*i], pat, j)))	logic error
27		#include <stdlib.h>	library inclusion
	return (c == LITCHAR || c == BOL || c == EOL || c == ANY	return (c == LITCHAR || c == BOL || c == ANY	logic error
	change(pat, sub); return 0;	funcB();atexit(funcB) ;void funcB(){change(pat, sub);}	added code
28	return (c == BOL || c == EOL ||c ==CLOSURE);	return (c == BOL || c == EOL || c == CLOSURE || c == CCL);	logic error
29	return (c == BOL || c == EOL ||c ==CLOSURE);	return (c == BOL || c == EOL || c == CLOSURE || c == NCCL);	logic error
30	return (c == BOL || c == EOL ||c ==CLOSURE);	return (c == BOL || c == EOL || c == CLOSURE || c != LITCHAR);	logic error
31	if ((lin[*i] != NEWLINE)&& (!locate(lin[*i], pat, j+1)))	if ((lin[*i] >= NEWLINE)&& (!locate(lin[*i], pat, j+1)))	logic error
32	else if ((isalnum(src [*i - 1]))&& (isalnum(src[*i + 1]))	else if ((isalnum(src [*i - 1]))& (isalnum(src[*i + 1]))	logic error

Table H.4: Table of Schedule Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`for(i=1; f_ele && (i<n ); i++)`	`for (i=1; f_list-> first && (i<n); i++)`	logic error
2	`count = block_queue-> mem_count`	`count = block_queue-> mem_count + 1`	logic error
	`n = (int)(count*ratio +1)`	`n = (int)(count*ratio)`	logic error
3	`n = (int)(count*ratio +1);`	`n = (int)(count*ratio +1.1);`	logic error
4	`if(count > 0){`	`if (count > 1){`	logic error
5	`if (proc){`	`/* if (proc){ */`	commented line
	`}`	`/* } */`	commented line
6	`for (i=1; f_ele && (i< n); i++)`	`for (i=1; f_list && (i <n); i++)`	logic error
7		`if(ratio == 1.0)n--;`	added if condition
		`if(ratio == 1.0)n--;`	added if condition
8	`proc->priority = prio;`	`/* proc->priority = prio; */`	commented line
9	`if (argc<(MAXPRIO+1))`	`if (argc < (MAXPRIO))`	logic error

Table H.5: Table of Schedule2 Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`if(prio < 1 || prio > MAXLOPRIO)return( BADPRIO);`	`/* if(prio < 1 || prio > MAXLOPRIO) return(BADPRIO); */`	commented lines
2	`index = index >= length ? length -1 : index;`	`/* index = index >= length ? length -1 : index;*/`	commented line
3	`if(ratio < 0.0 || ratio > 1.0)return( BADRATIO);`	`/* if(ratio < 0.0 || ratio > 1.0)return( BADRATIO); */`	commented lines
5		`if(prio < 1)return( BADPRIO);`	added if condition
6	`*prio = *command = -1; *ratio =-1.0;`	`*prio = 1; *command = -1; *ratio =1.0;`	logic error
7	`if(ratio < 0.0 || ratio > 1.0)return( BADRATIO);`	`if(ratio < 0.0 || ratio >= 1.0)return( BADRATIO);`	logic error

Continued on next page

Table H.5 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
8	if(prio > MAXPRIO || prio < 0)return(BADPRIO);	/* if(prio > MAXPRIO || prio < 0)return(BADPRIO); */	commented lines
9	reschedule(0);	get_current();	new changes
10	if(status =put_end(prio, new_process)) return(status);	put_end(prio, new_process);	removed if condition

Table H.6: Table of Tcas Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	result = !(Own_Below_Threat())|| ((Own_Below_Threat())&& (!(Down_Separation >= ALIM ())));	result = !(Own_Below_Threat())|| ((Own_Below_Threat())&& (!(Down_Separation > ALIM ())));	logic error
2	return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);	return (Climb_Inhibit ? Up_Separation + MINSEP : Up_Separation);	logic error
3	&& Other_RAC == NO_INTENT;	|| Other_RAC == NO_INTENT;	logic error
4	result = Own_Above_Threat ()&& (Cur_Vertical_Sep >= MINSEP)&& (Up_Separation >= ALIM());	result = Own_Above_Threat ()&& (Cur_Vertical_Sep >= MINSEP)|| (Up_Separation >= ALIM());	logic error
5	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)&& (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);	logic error
6	return (Own_Tracked_Alt < Other_Tracked_Alt);	return (Own_Tracked_Alt <= Other_Tracked_Alt);	logic error
7	Positive_RA_Alt_Thresh[1] = 500;	Positive_RA_Alt_Thresh[1] = 550;	logic error
8	Positive_RA_Alt_Thresh[3] = 740;	Positive_RA_Alt_Thresh[3] = 700;	logic error
9	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;	logic error
10	return (Own_Tracked_Alt < Other_Tracked_Alt);	return (Own_Tracked_Alt <= Other_Tracked_Alt);	logic error
	return (Other_Tracked_Alt < Own_Tracked_Alt);	return (Other_Tracked_Alt <= Own_Tracked_Alt);	logic error

Continued on next page

Table H.6 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
11	return (Own_Tracked_Alt < Other_Tracked_Alt);	return (Own_Tracked_Alt <= Other_Tracked_Alt);	logic error
	return (Other_Tracked_Alt < Own_Tracked_Alt);	return (Other_Tracked_Alt <= Own_Tracked_Alt);	logic error
	if (need_upward_RA && need_downward_RA)alt_sep = UNRESOLVED; else if (need_upward_RA)	if (need_upward_RA)	removed code
12	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)&& (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence || (Own_Tracked_Alt_Rate <= OLEV)&& (Cur_Vertical_Sep > MAXALTDIFF);	logic error
13	#define OLEV 600	#define OLEV 600+100	#define constant mutation
14	#define MAXALTDIFF 600	#define MAXALTDIFF 600+50	#define constant mutation
15	enabled = High_Confidence &&(Own_Tracked_Alt_Rate <= OLEV)&&(Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);	logic error
16	Positive_RA_Alt_Thresh[0] = 400;	Positive_RA_Alt_Thresh[0] = 400+1;	logic error
17	Positive_RA_Alt_Thresh[1] = 500;	Positive_RA_Alt_Thresh[1] = 500+1;	logic error
18	Positive_RA_Alt_Thresh[2] = 640;	Positive_RA_Alt_Thresh[2] = 640+50;	logic error
19	Positive_RA_Alt_Thresh[3] = 740;	Positive_RA_Alt_Thresh[3] = 740+20;	logic error
20	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;	logic error
21	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = (Up_Separation + NOZCROSS)> Down_Separation;	logic error
22	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = Up_Separation > Down_Separation;	logic error
23	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = (Up_Separation + NOZCROSS)> Down_Separation;	logic error

Continued on next page

Table H.6 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
24	upward_preferred = Inhibit_Biased_Climb()> Down_Separation;	upward_preferred = Up_Separation > Down_Separation;	logic error
25	result = !(Own_Above_Threat())|| ((Own_Above_Threat())&& (Up_Separation >= ALIM()));	result = !(Own_Above_Threat())|| ((Own_Above_Threat())&& (Up_Separation > ALIM()));	logic error
26	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)&& (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Cur_Vertical_Sep > MAXALTDIFF);	logic error
27	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)&& (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);	logic error
28	return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);	return ((Climb_Inhibit == 0)? Up_Separation + NOZCROSS : Up_Separation);	logic error
29	return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);	return (Up_Separation);	logic error
30	return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);	return (Up_Separation + NOZCROSS);	logic error
31		result = result && (Own_Tracked_Alt <= Other_Tracked_Alt);	added code
		result = result && (Own_Tracked_Alt < Other_Tracked_Alt);	added code
	need_upward_RA = Non_Crossing_Biased_Climb ()&& Own_Below_Threat();	need_upward_RA = Non_Crossing_Biased_Climb ();	logic error
32	need_downward_RA = Non_Crossing_Biased_Descend ()&& Own_Above_Threat();	need_downward_RA = Non_Crossing_Biased_Descend ();	logic error

Continued on next page

Table H.6 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
33	Positive_RA_Alt_Thresh [0] = 400; Positive_RA_Alt_Thresh [1] = 500; Positive_RA_Alt_Thresh [2] =640; Positive_RA_Alt_Thresh [3] = 740;	Positive_RA_Alt_Thresh [1] = 400; Positive_RA_Alt_Thresh [2] = 500; Positive_RA_Alt_Thresh [3] = 640; Positive_RA_Alt_Thresh [4] = 740;	logic error
34	if (enabled && ((tcas_equipped && intent_not_known)|| ! tcas_equipped))	if (enabled && tcas_equipped && intent_not_known || ! tcas_equipped)	logic error
35	return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);	return (Climb_Inhibit ? Up_Separation : Up_Separation + NOZCROSS);	logic error
36	#define DOWNWARD_RA 2	#define DOWNWARD_RA 1	#define constant mutation
37	return Positive_RA_Alt_Thresh[Alt_Layer_Value];	return Positive_RA_Alt_Thresh [0];	logic error
38	int Positive_RA_Alt_Thresh [4];	int Positive_RA_Alt_Thresh [3];	logic error
39	result = !(Own_Above_Threat())|| ((Own_Above_Threat())&& (Up_Separation >= ALIM()));	result = !(Own_Above_Threat())|| ((Own_Above_Threat())&& (Up_Separation > ALIM()));	logic error
40	result = !(Own_Below_Threat())|| ((Own_Below_Threat())&& (!(Down_Separation >= ALIM ())));	result = ((Own_Below_Threat())&& (!(Down_Separation >= ALIM ())));	logic error
	need_upward_RA = Non_Crossing_Biased_Climb ()&& Own_Below_Threat();	need_upward_RA = Non_Crossing_Biased_Climb ();	logic error
41	result = Own_Above_Threat ()&& (Cur_Vertical_Sep >= MINSEP)&& (Up_Separation >= ALIM());	result = (Cur_Vertical_Sep >= MINSEP)&& (Up_Separation >= ALIM());	logic error

Table H.7: Table of Tot_Info Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	goto ret1;	/* goto ret1; */	commented line
2	if (scanf(" %ld", &x (i,j))!= 1)	if (scanf(" %ld", &x (i,j))== 0)	logic error
3	if (r * c > MAXTBL)	if (r * c > MAXTBL -10)	logic error
4	if (Abs(gold)< EPS * Abs(g))	if (Abs(gold)< Abs(g))	logic error
5	totinfo += info;	totinfo = info;	logic error
6	#define MAXLINE 256	#define MAXLINE 56	#define constant mutation
7	if (pi > 0.0)	if (pi >= 0.0)	logic error
8	if (Abs(del)< Abs(sum)* EPS)return sum * exp(-x + a * log(x) - LGamma(a));	if (Abs(del)< Abs(sum)* EPS)return sum * exp(x + a * log(x)- LGamma(a));	logic error
9	totdf += infodf;	totdf = infodf;	logic error
10	double N;	float N;	data type changes
11	sum += del *= x/++ap;	sum = del *= x/++ap;	logic error
12	return -tmp + log(2.50662827465 * ser);	return -tmp + log(2.50663 * ser);	logic error
13	if (pj > 0.0)	if (pj >= 0.0)	logic error
14	if (r * c > MAXTBL)	if (r * c >= MAXTBL)	logic error
15	if (Abs(del)< Abs(sum)* EPS)	if (Abs(del)< Abs(sum)* (EPS-.000001))	logic error
16	if (info >= 0.0)	if (info >= 0.1)	logic error
17	anf = an * fac;	anf = an - fac;	logic error
18	if (rdf <= 0 || cdf <= 0)	if (rdf == 0 || cdf == 0)	logic error
19	#define MAXLINE 256	#define MAXLINE 26	#define constant mutation
20	if (rdf <= 0 || cdf <= 0)	if (rdf <= 0)	logic error

Continued on next page

Table H.7 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
21	#define MAXTBL 1000	#define MAXTBL 5000	#define constant mutation
22	if (N <= 0.0)	if (N <= 1.0)	logic error
23	for (n = 1; n <= ITMAX; ++n)	for (n = 0; n <= ITMAX; ++n)	logic error

Table H.8: Table of Cal Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	if(argc == 2)	if(argc = 2)	logic error
2	y = tm->tm_year + 1900;	y = tm->tm_year + 1800;	logic error
3	if(y<1 || y>9999){	if(y<1 || y<9999){	logic error
4	for(i=0; i<6*24; i +=24)	for(i=0; i<=6*24; i +=24)	logic error
5	if(y<1 || y>9999){	if(y>9999){	logic error
6	for(i=0; i<12; i+=3){	for(i=0; i<12; i+=4){	logic error
7	if(*s++ == '\0')	if(*++s == '\0')	logic error
8	if(*--s != ' ')	if(*--s == ' ')	logic error
9	30, 31, 30, 31,	30, 31, 30, 30,	logic error
10	mon[2] = 29;	mon[3] = 29;	logic error
11	mon[9] = 30;	mon[9] = 31;	logic error
12	case 1:mon[2] = 28; break;		removed switch case statement
13	if(i==3 && mon[m]==19)	if(i==3 || mon[m]==19)	logic error
14	s++;		removed statement
15	d %= 7;	i %= 7;	logic error
16	s = p+w;	s = p;	logic error

Continued on next page

Table H.8 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
17	`d = 4+y+(y+3)/4;`	`d = 4+y+(y*3)/4;`	logic error
18	`d += (y-1601)/400;`	`d += y-1601/400;`	logic error
19	`d += 3;`	`d = 3;`	logic error
20	`cal(i+3, y, string+46, 72);`	`cal(i+3, j, string+46, 72);`	logic error

Table H.9: Table of Checkeq Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`if (argc <= 1)`	`if (argc <= 0)`	logic error
2	`while (--argc > 0){`	`while (argc-- > 0){`	logic error
3	`if ((fin = fopen(*++ argv, "r"))== NULL){`	`if ((fin = fopen(*argv ++, "r"))== NULL){`	logic error
4	`start = 0;`	`totdel = 0;`	variable changes
5	`start = eq = line = ndel = totdel = 0;`	`start = line = ndel = totdel = 0;`	added variable
7	`ndel = 0;`	`ndel = 1;`	logic error
8	`ndel++;`	`ndel--;`	logic error
9	`totdel = 0;`		removed statement
10	`if (*in=='.' && *(in +1)=='E' && *(in+2)=='Q'){`	`if (*in=='.' && *(in +1)=='E' || *(in+2)=='Q'){`	logic error
11	`} else if (*in=='.' && *(in+1)=='E' && *(in +2)=='N'){`	`} else if (*in=='.' && *(in+1)=='E' || *(in +2)=='N'){`	logic error
12	`eq = 0;`	`eq = 1;`	logic error
14	`delim = *p;`	`line = *p;`	variable changes
15	`delim = *p;`	`delim = *(++p);`	logic error
16	`else printf(" New delims %c%c, line %d\n ", delim, delim, line)`		removed else condition
17	`if (ndel>0 && eq>0)`	`if (ndel>0 || eq>0)`	logic error

Continued on next page

Table H.9 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
18	`if (ndel > 0 && eq > 0)`	`if (ndel > 0 && eq <= 0)`	logic error
19	`totdel += ndel;`	`totdel -= ndel;`	logic error
20	`else {printf(" %d line %c%c, lines %d-%d\n", line-start+1, delim, delim, start, line); start = line;}`		removed else condition
21	`if (start > 0){`	`if (start== 0){`	logic error
22	`if (totdel)`	`if (ndel)`	variable changes

Table H.10: Table of Col Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`#define GREEK 0200`	`#define GREEK 0x200`	#define constant mutation
2	`fflag++;`	`hflag++;`	variable changes
3	`pgmname = argv[0];`	`pgmname = argv[1];`	logic error
5	`break;`		removed break statement
6	`for (p = argv[i]+1;*p; p++){`	`for (p = argv[i]-1;*p; p++){`	logic error
8	`pcp = 0;`	`pcp = 1;`	logic error
9	`cp = 0;`	`lp = 0;`	variable changes
11	`if (--half < -1){`	`if (--half < -1)`	logic error
14	`greek = GREEK;`	`greek = greek;`	logic error
15	`cp = (cp + 8)& -8;`	`cp = cp + (8 & -8);`	logic error
16	`if (cp > 0)`	`if (cp >= 0)`	logic error
17	`cp = 0;`	`lp = 0;`	variable changes
18	`c &= 0177;`	`c |= 0177;`	logic error

Continued on next page

Table H.10 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
20	`outc(c | greek);`	`outc(c || greek);`	logic error
21	`if (*--p) !`	`if (*p--)!`	logic error
22	`lp--;`	`lp++;`	logic error
23	`if (bflag || *line == '\0' || *line == ' ')`	`if (bflag || *line == '\0' )`	logic error
24	`c1 = *++line;`	`c1 = ++(*line);`	logic error
26	`c1=c2; c2=c3;`	`c2=c3; c1=c2;`	logic error
27	`if (page[lno] == 0){`	`if (page[lno] = 0){`	logic error
28	`lno %= PL;`	`lno /= PL;`	logic error
29	`static int cline = 0;`	`int cline = 0;`	data type changes
30	`while (cline < lineno - 1){`	`while (cline < lineno) {`	logic error
31	`store (ll--);`	`store (--ll);`	logic error
32	`ncp = pcp;`	`pcp = ncp;`	logic error
33	`if ((++ncp & 7)== 0 && hflag){`	`if ((++ncp | 7)== 0 && hflag){`	logic error
34	`if (!*--p)`	`if (*--p)`	logic error
35	`if (gflag != (*p & GREEK)&& *p != '\b'){`	`if (gflag != (*p & GREEK)|| *p != '\b'){`	logic error
36	`gflag ^= GREEK;`	`hflag ^= GREEK;`	variable changes
37	`putchar (*p & ~GREEK);`	`putchar(*p && ~GREEK);`	logic error

Table H.11: Table of Spline Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`else if(!getfloat(&x.val[n]))break;`		removed else condition
4	`for(j=m;j>0||i==0&&j==0;j--){`	`for(j=m;j>0&&i==0&&j==0;j--){`	logic error
5	`d = 1;`	`d = 0;`	logic error

Continued on next page

Table H.11 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
6	m = 1.001*m*hi1/(x.ub-x.lb);	m = 0.001*m*hi1/(x.ub-x.lb);	logic error
7	yy =D2yi*(x0-x0*x0*x0) +D2yi1*(x1-x1*x1*x1);	yy =D2yi*(x0-x0*x0*x0) +D2yi1*x1-x1*x1*x1;	logic error
9	x.val[i+1]-x.val[i];	x.val[i+1]-x.val[0];	logic error
11	zz = (y.val[i]-y.val[i -1])/(x.val[i]-x.val[i -1]);	zz = (x.val[i]-x.val[i -1])/(x.val[i]-x.val[i -1]);	logic error
12	r[i] = rhs(i)-hi*r[i -1]/d;	r[i] = rhs(i)+hi*r[i -1]/d;	logic error
13	a - hi*hi/d;	a - hi*hi*d;	logic error
14	if(i==0)D2yi = konst* D2yi1;	if(i=0)D2yi = konst* D2yi1;	logic error
15	hi = x.val[i]-x.val[i -1];	hi = y.val[i]-x.val[i -1];	logic error
16	x1 = j*h/hi1;	x1 = j*h;	logic error
17	s = -hi*s/d;		removed statement

Table H.12: Table of Tr Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	else if(dflag)code[i] = 0;	else if(cflag)code[i] = 0;	variable changes
2	if(!sflag || c!=save || !squeez[c&0377]){	if(!dflag || c!=save || !squeez[c&0377]){	logic error
3	string1.last = string2 .last = 0;	string1.last = string2 .last = 1;	logic error
5	case 's':sflag++; continue;		removed switch case statement
9	if(cflag)c = *compl++;	if(cflag)c = ++*compl;	logic error
10	else lastd = d;		removed else condition
11	code[c&0377] = dflag ?1:d;	code[c&0377] = dflag?d :1;	logic error
12	squeez[d&0377] = 1;	squeez[d|0377] = 1;	logic error

Continued on next page

Table H.12 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
14	`if(s->last && *s->p =='-'){`	`if(*s->p=='-'){`	logic error
16	`n = n*8 + c - '0';`	`n = n*8 + c + '0';`	logic error
17	`compl = vect;`	`compl = code;`	variable changes

Table H.13: Table of Uniq Bug Information

Ver	Base Version	Seeded Version	Bug Type
1	`else mode = argv [1][1];`		removed else condition
2	`linec++;`	`linec--;`	logic error
4	`*buf++ = c;`	`*++buf = c;`	logic error
5	`return;`		removed return statement
6	`uniq = 0;`	`uniq = 1;`	logic error
7	`while((c = *b1++)= 0) !`	`while((c = *b1--)= 0)!`	logic error
8	`while(nf++<fields){`	`while(nf++ <= fields){`	logic error
9	`while( !(*s == ' ' || *s == '\t' || *s == 0))`	`while( !(*s == ' ' || *s == '\t' && *s == 0))`	logic error
10	`while(nl++ < letters && *s!= 0)`	`while(nl++ < letters && s!=0)`	logic error
12	`if(*b2 = 0) !`	`if(*b1 = 0)!`	logic error
13	`while((c = *b1++)= 0) !`	`while((c = *++b1)= 0)!`	logic error
14	`while( !(*s == ' ' || *s == '\t' || *s == 0))`	`while( !(*s == ' ' || *s == '\t'))`	logic error
15	`while( !(*s == ' ' || *s == '\t' || *s == 0))`	`while( !(*s == ' ' || *s == '\t')|| *s == 0 )`	logic error
16	`linec = 0;`	`linec = 1;`	logic error
17	`return(s);`	`return(++s);`	logic error

Continued on next page

Table H.13 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
18	`uniq = 0;`	`linec = 0;`	variable changes
19	`break;`		removed break statement

Table H.14: Table of Concordance Bug Information

Ver	Correct Version	Buggy Version	Description
1	`assert (fp);`		added assert statement
2	`if (head){`		added if condition
3	`int wordBfrSize = MAXBFRSZ;`	`char wordBfr[MAXBFRSZ ];`	added code
	`wordBfr = (char *) malloc (sizeof (char)* wordBfrSize);`		
4	`assert (confile != 0);`	`if (confile == 0){ cerr << "Couldn't open concordance file " << conFn << endl;perror ("File opening error") ;exit(1); }`	added assert statement
	`assert (alphafile!=0);`	`if (alphafile == 0){ cerr << "Couldn't open alphabet file " << alphaFn << endl;perror ("File opening error ");exit(1);}`	added assert statement
5	`else if (inbuffer[n] == '>' && isNum)`	`else if (inbuffer[n] == '>' && countType == 's' && isNum)`	logic error
6	`for (n = 1; argv[1][ n]; n++){if (!strchr ("slpqn:0123456789", argv[1][n])){cerr << " \n\nSYNTAX ERROR on command line: invalid switch character\n"; exit(1);}}`		added for-loop
7	`if (argc < 2 || argc > 4 || ((argc == 4)&& ( argv[0][1] != '-')))`	`if (argc < 2)`	logic error

Continued on next page

Table H.14 -- continued from previous page

Ver	Correct Version	Buggy Version	Description
8	else if (argc == 2 && argv[1][0] == '-'){ instructions ();return 0;}		added elseif statement
9	if (argc == 2 || (argc == 3 && argv[1][0] == '-'))	if (argc < 4)	logic error
	outfile = new char[strlen (argv[argc - 1])];	outfile = new char[strlen (argv[3])];	logic error
	strcpy (outfile, argv [argc - 1]);	strcpy (outfile, argv [3]);	logic error
	if (argc > 2 && argv [1][0] == '-')	if (argc > 2)	logic error
10	n = 1;	for (n = 1; n < argc; n++){if (strstr (argv[n], "-")){	removed for-loop
11	if (argv[1][n] != 0) {cerr <<''\n\nSYNTAX ERROR on command line: switch -n: number too big\n'';exit(1);}		added if condition
12	for (n = 0; argv[1][n] != 'n'; n++);if (argv [1][n+1] != ':')	for (n = 0; argv[1][n] != ':'; n++); n++	logic error
13	if (n == 0){cerr <<''\ n\n SYNTAX ERROR on command line: switch -[pvl]n: did not contain a proper integer.\n'';exit(1);}		added if condition

Table H.15: Table of Space Bug Information

Ver	Base Version	Seeded Version	Bug Type
3	*curr_ptr = *tp;	curr_ptr = tp;	removed pointer variable
4	*pp2 = *curr_ptr;	pp2 = curr_ptr;	removed pointer variable
5	*pp2 = *curr_ptr;	pp2 = curr_ptr;	removed pointer variable

Continued on next page

Table H.15 -- continued from previous page

Ver	Base Version	Seeded Version	Bug Type
6	*tp = *curr_ptr;	tp = curr_ptr;	removed pointer variable
14	error = (GetKeyword (Keywords[88], curr_ptr));	GetKeyword (Keywords [88], curr_ptr);	removed variable
15	error = (GetKeyword (Keywords[88], curr_ptr));	GetKeyword (Keywords [88], curr_ptr);	removed variable
17	port_ptr->OMIT_POL = YES;		removed statement
18	app_ptr->PHEA += phi;	app_ptr->PSEA += phi;	variable changes
19		if ((group_ptr ->ELEM_PTR)-> POLARIZATION != LIN_POL)return 0;	added if condition
20	can += angle_step;	can = angle_step;	logic error
21	cph += phase_step;	cph = phase_step;	logic error
23	while (app_ptr!=NULL)	while (app_ptr->NEXT != NULL)	pointer variable changes
28	*app_ptr =*addrem_ptr;	app_ptr = addrem_ptr;	removed pointer variables
31	if (error == 17)	if (error != 0)	logic error
33	gnode_ptr->PHEA += angle;	gnode_ptr->PHEA = angle;	logic error

University Library

A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Lee, Hua Jie

Title:
Software debugging using program spectra

Date:
2011

Citation:
Lee, H. J. (2011). Software debugging using program spectra. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

Persistent Link:
http://hdl.handle.net/11343/36410

File Description:
Software debugging using program spectra

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.